Abstract Learning Frameworks for Synthesis

Christof Löding[1], P. Madhusudan[2], and Daniel Neider[2,3]

[1] RWTH Aachen
[2] University of Illinois, Urbana-Champaign
[3] University of California, Los Angeles

Abstract. We develop abstract learning frameworks for synthesis that embody the principles of the CEGIS (counterexample-guided inductive synthesis) algorithms in current literature. Our framework is based on iterative learning from a hypothesis space that captures synthesized objects, using counterexamples from an abstract sample space, and a concept space that abstractly defines the semantics of synthesis. We show that a variety of synthesis algorithms in current literature can be embedded in this general framework. We also exhibit three general recipes for convergent synthesis: the first two recipes based on finite spaces and Occam learners generalize all techniques of convergence used in existing engines, while the third, involving well-founded quasi-orderings, is new, and we instantiate it to concrete synthesis problems.

1 Introduction

The field of synthesis, which includes several forms of synthesis including synthesizing controllers [37], program expressions [43], program repairs [26], program translations [11,23], loop invariants [17,16], and even entire programs [32,25], has become a fundamental and vibrant subfield in programming languages. While classical studies of synthesis have focused on synthesizing entire programs or controllers from specifications [32,37], there is a surge of tractable methods that have emerged in recent years in synthesizing small program expressions. These expressions often are complex but small, and are applicable in niche domains such as program sketching [43] (finding program expressions that complete code), synthesizing Excel programs for string transformations [18], synthesizing super-optimized code [40], deobfuscating code [21], synthesizing invariants to help in verification [16,17], etc.

One prominent technique that has emerged in recent years for expression synthesis is based on *inductively learning expressions from samples*. Assume the synthesis problem is to synthesize an expression e that satisfies some specification $\psi(e)$. The crux of this approach is to *ignore* the precise specification ψ, and instead synthesize an expression based on certain *facets* of the specification. These incomplete facets of the specification are often much simpler in structure and in logical complexity compared to the specification, and hence synthesizing an expression satisfying the constraints the facets impose is more tractable. The learning-based approach to synthesis hence happens in rounds— in each round,

the learner synthesizes an expression that satisfies the current facets, and a *verification oracle* checks whether the expression satisfies the actual specification ψ, and if not, finds a new facet of the specification witnessing this. The learner then continues to synthesize by adding this new facet to its collection.

This counter-example guided inductive synthesis (CEGIS) approach [42] to synthesis in current literature philosophically advocates precisely this kind of inductive synthesis. The CEGIS approach has emerged as a powerful technique in several domains of both program synthesis as well as program verification ranging from synthesizing program invariants for verification [16,17] to specification mining [3], program expressions that complete sketches [43], superoptimization [40], control [22], string transformers for spreadsheets [18], protocols [45], etc.

The goal of this paper is to develop a *theory of iterative learning-based synthesis* through a formalism we call *abstract learning frameworks for synthesis*. The framework we develop aims to be general and abstract, encompassing several known CEGIS frameworks as well as several other synthesis algorithms not generally viewed as CEGIS. The goal of this line of work is to build a framework, with accompanying concepts, definitions, and vocabulary that can be used to understand and combine learning-based synthesis across different domains.

An abstract learning framework (ALF) (see Figure 1 on Page 5) consists of three spaces: $\mathcal{H}$, $\mathcal{S}$, and $\mathcal{C}$. The *(semantic) concept space* $\mathcal{C}$ gives semantic descriptions of the concepts that we wish to synthesize, the *hypotheses space* $\mathcal{H}$ comprises restricted (typically syntactically restricted) forms of the concepts to synthesize, and the sample space $\mathcal{S}$ consists of samples (modeling facets of the specification) from which the learner synthesizes hypotheses. The spaces $\mathcal{H}$ and $\mathcal{S}$ are related by a variety of functions that give semantics to samples and semantics to hypotheses using the space $\mathcal{C}$. The conditions imposed on these relations capture the learning problem precisely, and their abstract formulation facilitates modeling a variety of synthesis frameworks in the literature.

The target for synthesis is specified as a *set* of semantic concepts. This is an important digression from classical learning frameworks, where often one can assume that there is a *particular* target concept that the learner is trying to learn. Note that in synthesis problems, we must *implement the teacher as well*, and hence the modeling of the target space is important. In synthesis problems, the teacher does not have a single target in mind nor does she know explicitly the target set (if she knew, there would be no reason to synthesize!). Rather, she knows the *properties* that capture the set of target concepts. For instance, in invariant synthesis, the teacher knows the properties of a set being an invariant for a loop, and this defines implicitly a *set* of invariants as target. The teacher needs to examine a hypothesis and check whether it satisfies the properties defining the target set. Consequently, we can view the teacher as a *verification oracle* that checks whether a hypothesis belongs to the implicitly defined target set.

We exhibit a variety of existing synthesis frameworks that can be naturally seen as instantiations of our abstract-learning framework, where the formulation shows the diversity in the instantiations of the spaces. These include (a) a variety of CEGIS-based synthesis techniques for synthesizing program expressions in

sketches (completing program sketches [43], synthesizing loop-free programs [19], mining specifications [22], synthesizing synchronization code for concurrent programs [10], etc.), (b) synthesis from input-output examples such as Flashfill [18], (c) the CEGIS framework applied to the concrete problem of solving synthesis problems expressed in the SMT-based SyGuS format [2,1], and three synthesis engines that use learning to synthesize solutions, (d) invariant synthesis frameworks, including Houdini [14] and the more recent ICE-learning model for synthesizing loop invariants [16], spanning a variety of domains from arithmetic [16,17] to quantified invariants over data structures [15], and (e) synthesizing fixed-points and abstract transformers in abstract interpretation settings [44].

Formalizing of synthesis algorithms as ALFs can help highlight the nuances of different learning-based synthesis algorithms, even for the *same* problem. One example comprises two inductive learning approaches for synthesizing program invariants— one based on the ICE learning model [16], and the second which is any synthesis engine for logically specified synthesis problems in the SyGuS format, which can express invariant synthesis. Though both can be seen as CEGIS-based synthesis algorithms, the sample space for them are very different, and hence the synthesis algorithms are also different— the significant performance differences between SyGuS-based solvers and ICE-based solvers(the latter performing better) in the recent SyGuS competition (invariant-synthesis track) suggest that this choice may be crucial [4]. Another example are two classes of CEGIS-based solvers for synthesizing linear integer arithmetic functions against SyGuS specifications— one based on a sample space that involves purely inputs to the function being synthesized [39,35], while the other is the more standard CEGIS algorithm based on valuations of quantified variables.

We believe that just describing an approach as a learning-based synthesis algorithm or a CEGIS algorithm does not convey the nuances of the approach— it is important to precisely spell out the sample space and the semantics of this space with respect to the space of hypotheses being learned. The ALF framework gives the vocabulary in phrasing these nuances, allowing us to compare and contrast different approaches.

Convergence: The second main contribution of this paper is to study *convergence* issues in the general abstract learning-based framework for synthesis. We first show that under the reasonable assumptions that the learner is consistent (always proposes a hypothesis consistent with the samples it has received) and the teacher is honest (gives a sample that distinguishes the current hypothesis from the target set without ruling out any of the target concepts), the iterative learning will always converge *in the limit* (though, not necessarily in finite time, of course). This theorem vouches for the correctness of our abstract formalism in capturing abstract learning, and utilizes all the properties that define ALFs.

We then turn to studying strategies for convergence in finite time. We propose three general techniques for ensuring successful termination for the learner. First, when the hypothesis space is bounded, it is easy to show that any consistent learner (paired with an honest teacher) will converge in finite time. Several examples of these exist in learning— learning conjunctions as in the Houdini algorithm [14],

etc., learning Boolean functions (like decision-tree learning with purely Boolean predicates as attributes) or functions over bit-vector domains (Sketch [43] and the SyGuS solvers that work on bit-vectors), and learning invariants using specialized forms of a finite class of automata that capture list/array invariants [15].

The second recipe is a formulation of the Occam's razor principle that uses parsimony/simplicity as the learning bias [6]. The idea of using Occam's principle in learning is prevalent (see Chapter 2 of [24] and [33]) though its universal appeal in generalizing concepts is debatable [13]. We show, however, that learning using Occam's principle helps in convergence. A learner is said to be an Occam learner if there is a *complexity ordering*, which needs to be a total quasi order where the set of elements below any element is finite, such that the learner always learns *a smallest* concept according to this order that is consistent with the sample. We can then show that any Occam learner will converge to some target concept, if one exists, in finite time. This result generalizes many convergent learning mechanisms that we know of in the literature (for example, the convergent ICE-learning algorithms for synthesizing invariants using constraint solvers [16], and the enumerative solvers in almost every domain of synthesis [28,26,36,45], including for SyGuS [2,1], that enumerate by dovetailing through expressions).

The first two recipes for finite convergence cover all the methods we know in the literature for convergent learning-based synthesis, to the best of our knowledge. The third recipe for finite convergence is a more complex one based on well-founded quasi orderings. This recipe is involved and calls for using clever initial queries that force the teacher to divulge information that then makes the learning space tractable. We do not know of any existing synthesis learning frameworks that use this natural recipe, but propose two new convergent learning algorithms following this recipe, one for intervals, and the other for conjunctive linear inequality constraints over a set of numerical attributes over integers.

2 Abstract Learning Frameworks for Synthesis

In this section we introduce our abstract learning framework for synthesis. Figure 1 gives an overview of the components and their relations that are introduced in the following (ignore the target $\mathcal{T}$, $\gamma^{-1}(\mathcal{T})$, and the maps τ and λ for now). We explain these components in more detail after the formal definition.

Definition 1 (Abstract Learning Frameworks). *An* abstract learning framework for synthesis *(ALF, for short), is a tuple* $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$, *with*

- *A class* $\mathcal{C}$*, called the concept space,*
- *A class* $\mathcal{H}$*, called the hypothesis space,*
- *A class* $\mathcal{S}$*, called the sample space, with a join semi-lattice* $(\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s)$ *defined over it,*
- *A concretization function* $\gamma : \mathcal{H} \to \mathcal{C}$*, and*
- *A consistency function:* $\kappa : \mathcal{S} \to 2^{\mathcal{C}}$ *satisfying* $\kappa(\bot_s) = \mathcal{C}$ *and* $\kappa(S_1 \sqcup S_2) = \kappa(S_1) \cap \kappa(S_2)$ *for all* $S_1, S_2 \in \mathcal{S}$*. If the second condition is relaxed to* $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$*, we speak of a* general ALF*.*

We say an ALF has a complete sample space if the sample space $(\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s)$ *is a complete join semi-lattice (i.e., if the join is defined for arbitrary subsets of* $\mathcal{S}$*). In this case, the consistency relation has to satisfy* $\kappa(\bigsqcup(\mathcal{S}')) = \bigcap_{S \in \mathcal{S}'} \kappa(S)$ *for each* $\mathcal{S}' \subseteq \mathcal{S}$ *(and* $\kappa(\bigsqcup(\mathcal{S}')) \subseteq \bigcap_{S \in \mathcal{S}'} \kappa(S)$ *for general ALFs).*

As in computational learning theory, as presented in e.g., in [5] or [24], we consider a *concept space* $\mathcal{C}$, which contains the objects that we are interested in. For example, in an invariant synthesis setting in verification, an element $C \in \mathcal{C}$ would be a *set* of program configurations. In the synthesis setting, $\mathcal{C}$ could contain the objects we would like to synthesize, such as all functions from $\mathbb{Z}^n$ to $\mathbb{Z}$.

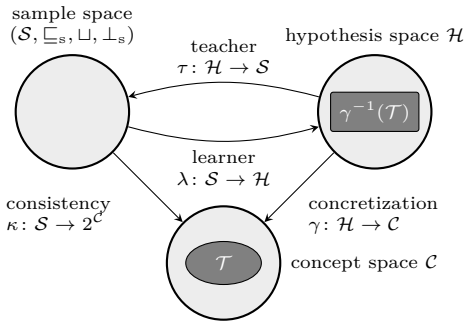

Fig. 1. Components of an ALF

The *hypothesis space* $\mathcal{H}$ contains the objects that the learner produces. These are representations of (some) elements from the concept space. For example, if $\mathcal{C}$ consists of all functions from $\mathbb{Z}^n$ to $\mathbb{Z}$, then $\mathcal{H}$ could consist the set of all functions expressible in linear arithmetic.

The relation between hypotheses and concepts is given by a concretization function $\gamma : \mathcal{H} \to \mathcal{C}$ that maps hypotheses to concepts (their semantics).

In classical computational learning theory for classification [24,33], one often considers samples consisting of positive and negative examples. If learning is used to infer a target concept that is not uniquely defined but rather should satisfy certain properties, then samples consisting of positive and negative examples are sometimes not sufficient. As we will show later, samples can be quite complex (see Section 4 for such examples, including implication counterexamples and grounded formulas).

We work with a *sample space*, which is a bounded join-semilattice $(\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s)$ (i.e., $\sqsubseteq_s$ is a partial order over $\mathcal{S}$ with $\bot_s$ as the least element, and $\sqcup$ is the binary least upper-bound operator on $\mathcal{S}$ with respect to this ordering). An element $S \in \mathcal{S}$, when given by the teacher, intuitively, gives some information about a target specification. The join is used by the learner to combine the samples returned as feedback by the teacher during iterative learning. The least element $\bot_s$ corresponds to the empty sample. We encourage the reader to think of the join as the union of samples.

The *consistency relation* κ captures the semantics of samples with respect to the concept space by assigning to each sample S the set $\kappa(S)$ of concepts that are consistent with the sample. The first condition on κ says that all concepts are consistent with the empty sample $\bot_s$. The second condition says that the set of samples consistent with the join of two samples is precisely the set of concepts that is consistent with both the samples. Intuitively, this means that joining samples does not introduce new inconsistencies, and existing inconsistencies transfer to bigger samples. The condition that $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$ is natural, as

it says that if a concept is consistent with the join of two samples, then the concept must be consistent with both of them individually. The condition that $\kappa(S_1 \sqcup S_2) \supseteq \kappa(S_1) \cap \kappa(S_2)$ is debatable; it claims that samples when taken together cannot eliminate a concept that they couldn't eliminate individually. We therefore mention the notion of *general ALF* in Definition 1. However, we have not found any natural example that requires such a generalization, and therefore prefer to work with ALFs instead of general ALFs in the rest of the paper. In Definition 4, we comment on what needs to be adapted to make the results of the paper go through for general ALFs.

Some other auxiliary definitions we will need: We define $\kappa_{\mathcal{H}}(S) := \{H \in \mathcal{H} \mid \gamma(H) \in \kappa(S)\}$ to be the set of hypotheses that are consistent with S. For a sample $S \in \mathcal{S}$ we say that S *is realizable* if there exists a hypothesis that is consistent with S (i.e., $\kappa_{\mathcal{H}}(S) \neq \emptyset$).

ALF Instances and Learners An instance of a learning task for an ALF is given by a specification that defines target concepts. The goal is to infer a hypothesis whose semantics is such a target concept. In classical computational learning theory, this target is a unique concept. In applications for synthesis, however, there can be many possible target concepts, for example, all inductive invariants of a program loop.

Formally, a *target specification* is just a set $\mathcal{T} \subseteq \mathcal{C}$ of concepts. An ALF instance combines an ALF and a target specification:

Definition 2 (ALF Instance). *An ALF instance is a pair $(A, \mathcal{T})$ where $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$ is an ALF and $\mathcal{T} \subseteq \mathcal{C}$ is a target specification.*

The goal of learning-based synthesis is for the learner to synthesize *some* element $H \in \mathcal{H}$ such that $\gamma(H) \in \mathcal{T}$. Furthermore, the role of the teacher is to instruct the learner giving reasons why the hypothesis produced by the learner in the current round does not belong to the target set.

There is a subtle point here worth emphasizing. In synthesis frameworks, the teacher does not explicitly know the target space $\mathcal{T}$. Rather she knows a definition of the target space, and she can examine a hypothesis H and check whether it satisfies the properties required of the target set. For instance, when synthesizing an invariant for a program, the teacher knows the properties of the invariant (inductiveness, etc.) and gives counterexample samples based on failed properties.

We say that the target specification is *realizable by a hypothesis*, or simply realizable, if there is some $H \in \mathcal{H}$ with $\gamma(H) \in \mathcal{T}$. For a hypothesis $H \in \mathcal{H}$, we often write $H \in \mathcal{T}$ instead of $\gamma(H) \in \mathcal{T}$.

As in classical computational learning theory, we define a *learner* (see Figure 1) to be a function that maps samples to hypotheses, and a consistent learner to be a learner that only proposes consistent hypotheses for samples.

Definition 3. *A learner for an ALF $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$ is a map $\lambda : \mathcal{S} \to \mathcal{H}$ that assigns a hypothesis to every sample. A* consistent learner *is a learner λ with $\gamma(\lambda(S)) \in \kappa(S)$ for all realizable samples $S \in \mathcal{S}$.*

Iterative learning. In the iterative learning setting, the learner produces a hypothesis starting from some initial sample (e.g., $\bot_{\mathrm{s}}$). For each hypothesis provided by the learner that does not satisfy the target specification, a teacher (see Figure 1) provides feedback by returning a sample witnessing that the hypothesis does not satisfy the target specification.

Definition 4. *Let $(A, \mathcal{T})$ be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$, and $\mathcal{T} \subseteq \mathcal{C}$. A* teacher *for this ALF instance is a function $\tau : \mathcal{H} \to \mathcal{S}$ that satisfies the following two properties:*

i) Progress: *$\tau(H) = \bot_{\mathrm{s}}$ for each target element $H \in \mathcal{T}$, and $\gamma(H) \notin \kappa(\tau(H))$ for all $H \notin \mathcal{T}$, and*

ii) Honesty: *$\mathcal{T} \subseteq \kappa(\tau(H))$ for each $H \in \mathcal{H}$.*[4]

Firstly, progress says that if the hypothesis is in the target set, then the teacher must return the "empty" sample $\bot_{\mathrm{s}}$, signaling that the learner has learned a target; otherwise, the teacher must return a sample that rules out the current hypothesis. This ensures that a consistent learner can never propose the same hypothesis again, and hence makes progress. Secondly, honesty demands that the sample returned by the teacher is consistent with *all* target concepts. This ensures that the teacher does not eliminate any element of the target set arbitrarily.

When the learner and teacher interact iteratively, the learner produces a sequence of hypotheses, where in each round it proposes a hypothesis $\lambda(S)$ for the current sample $S \in \mathcal{S}$, and then adds the feedback $\tau(\lambda(S))$ of the teacher to obtain the new sample.

Definition 5. *Let $(A, \mathcal{T})$ be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$, and $\mathcal{T} \subseteq \mathcal{C}$. Let $\lambda : \mathcal{S} \to \mathcal{H}$ be a learner, and let $\tau : \mathcal{H} \to \mathcal{S}$ be a teacher. The combined behavior of the learner λ and teacher τ is the function $f_{\tau,\lambda} : \mathcal{S} \to \mathcal{S}$, where $f_{\tau,\lambda}(S) := S \sqcup \tau(\lambda(S))$.*

The sequence of hypotheses generated by the learner λ and teacher τ is the transfinite sequence $\langle S^{\alpha}_{\tau,\lambda} \mid \alpha \in \mathbb{O} \rangle$, where $\mathbb{O}$ denotes the class of all ordinals, obtained by iterative application of $f_{\tau,\lambda}$:

- *$S^{0}_{\tau,\lambda} := \bot_{\mathrm{s}}$;*
- *$S^{\alpha+1}_{\tau,\lambda} := f_{\tau,\lambda}(S^{\alpha}_{\tau,\lambda})$ for successor ordinals; and*
- *$S^{\alpha}_{\tau,\lambda} := \bigsqcup_{\beta<\alpha} S^{\beta}_{\tau,\lambda}$ for limit ordinals.*

If the sample lattice is not complete, the above definition is restricted to the first two items and yields a sequence indexed by natural numbers.

The following lemma states that the teacher's properties of progress and honesty transfer to the iterative setting for consistent learners if the target specification is realizable. A proof can be found in the extended paper [31].

[4] For general ALFs one has to require that the least upper bound of all samples returned by the teacher is consistent with all targets (and for non-complete sample lattices the least upper bound of all possible finite sets of samples returned by the teacher).

Lemma 1. *Let $\mathcal{T}$ be realizable, λ be a consistent learner, and τ be a teacher. If $\mathcal{S}$ is a complete sample lattice, then*

(a) the learner makes progress: for all $\alpha \in \mathbb{O}$, either $\kappa(S^{\alpha}_{\tau,\lambda}) \supsetneq \kappa(S^{\alpha+1}_{\tau,\lambda})$ and $\lambda(S^{\alpha}_{\tau,\lambda}) \notin \kappa(S^{\alpha+1}_{\tau,\lambda})$, or $\lambda(S^{\alpha}_{\tau,\lambda}) \in \mathcal{T}$, and
(b) the sample sequence is consistent with the target specification: $\mathcal{T} \subseteq \kappa(S^{\alpha}_{\tau,\lambda})$ for all $\alpha \in \mathbb{O}$.

If $\mathcal{S}$ is a non-complete sample lattice, then (a) and (b) hold for all $\alpha \in \mathbb{N}$.

We end with an example of an ALF. Consider the problem of synthesizing guarded affine functions that capture how a piece of code P behaves, as in program deobfuscation. Then the concept class could be all functions from $\mathbb{Z}^n$ to $\mathbb{Z}$, the hypothesis space would be the set of all expressions describing a guarded affine function (in some fixed syntax). The target set (as a subset of $\mathcal{C}$) would consist of a *single* function $\{f_t\}$, where f_t is the function computed by the program P. For any hypothesis function h, let us assume we can build a teacher who can compare h and P for equivalence, and, if they differ, return a counterexample of the form $(\boldsymbol{i}, o)$, which is a concrete input $\boldsymbol{i}$ on which h differs from P, and o is the output of P on $\boldsymbol{i}$. Then the sample space would consist of sets of such pairs (with union for join and empty set for $\bot_{\mathrm{s}}$). The set of functions consistent with a set of samples would be the those that map the inputs mentioned in the samples to their appropriate outputs. The iterative learning will then model the process of synthesis, using learning, a guarded affine function that is equivalent to P.

3 Convergence of iterative learning

In this section, we study convergence of the iterative learning process. We start with a general theorem on transfinite convergence (convergence in the limit) for complete sample lattices. We then turn to convergence in finite time and exhibit three recipes that guarantee convergence.

From Lemma 1 one can conclude that the transfinite sequence of hypotheses constructed by the learner converges to a target set (see the extended paper [31] for a proof).

Theorem 1. *Let $\mathcal{S}$ be a complete sample lattice, $\mathcal{T}$ be realizable, λ be a consistent learner, and τ be a teacher. Then there exists an ordinal α such that $\lambda(S^{\alpha}_{\tau,\lambda}) \in \mathcal{T}$.*

The above theorem ratifies the choice of our definitions, and the proof (relying on Lemma 1) crucially uses all aspects of our definitions (the honesty and progress properties of the teacher, the condition imposed on κ in an ALF, the notion of consistent learners, etc.).

Convergence in finite time is clearly the more desirable notion, and we propose tactics for designing learners that converge in finite time. For an ALF instance $(\mathcal{A}, \mathcal{T})$, we say that a learner λ *converges for a teacher* τ if there is an $n \in \mathbb{N}$ such that $\lambda(S^{n}_{\tau,\lambda}) \in \mathcal{T}$, which means that λ produces a target hypothesis after n steps. We say that λ converges if it converges for every teacher. We say that λ converges from a sample S in case the learning process starts from a sample $S \neq \bot_{\mathrm{s}}$ (i.e., if $S^{0}_{\lambda,\tau} = S$).

Finite hypothesis spaces We first note that if the hypothesis space (or the concept space) is finite, then any consistent learner converges: by Lemma 1, the learner always makes progress, and hence never proposes two hypotheses that correspond to the same concept. Consequently, the learner only produces a finite number of hypotheses before finding one that is in the target (or declare that no such hypothesis exists).

There are several synthesis engines using learning that use finite hypothesis spaces. For example, Houdini [14] is a learner of *conjunctions* over a fixed finite set of predicates and, hence, has a finite hypothesis space. Learning decision trees over purely Boolean attributes (not numerical) [38] is also convergent because of finite hypothesis spaces, and this extends to the ICE learning model as well [17]. Invariant generation for arrays and lists using *elastic QDAs* [15] also uses a convergence argument that relies on a finite hypothesis space.

Occam Learners We now discuss the most robust strategy we know for convergence, based on the Occam's razor principle. Occam's razor advocates parsimony or simplicity [6], that the simplest concept/theory that explains a set of observations is better, as a virtue in itself. There are several learning algorithms that use parsimony as a learning bias in machine learning (e.g., *pruning* in decision-tree learning [33]), though the general applicability of Occam's razor in machine learning as a sound means to generalize is debatable [13]. We now show that in *iterative* learning, following Occam's principle leads to convergence in finite time. However, the role of *simplicity* itself is not the technical reason for convergence, but that there is *some* ordering of concepts that biases the learning.

Enumerative learners are a good example of this. In enumerative learning, the learner enumerates hypotheses in some order, and always conjectures the first consistent hypothesis. In an iterative learning-based synthesis setting, such a learner always converges on some target concept, if one exists, in finite time.

Requiring a total order of the hypotheses is in some situations too strict. If, for example, the hypothesis space consists of deterministic finite automata (DFAs), we could build a learner that always produces a DFA with the smallest possible number of states that is consistent with the given sample. However, the relation $\preceq$ that compares DFAs w.r.t. their number of states is not an ordering because there are different DFAs with the same number of states.

In order to capture such situations, we work with a *total quasi-order* $\preceq$ on $\mathcal{H}$ instead of a total order. A quasi-order (also called preorder) is a transitive and reflexive relation. The relation being total means that $H \preceq H'$ or $H' \preceq H$ for all $H, H' \in \mathcal{H}$. The difference to an order relation is that $H \preceq H'$ and $H' \preceq H$ can hold in a quasi-order, even if $H \neq H'$.

In analogy to enumerations, we require that each hypothesis has only finitely many hypotheses "before" it w.r.t. $\preceq$, as expressed in the following definition.

Definition 6. *A* complexity ordering *is a total quasi-order $\preceq$ such that for each $x \in \mathcal{H}$ the set $\{y \in \mathcal{H} \mid y \preceq x\}$ is finite.*

The example of comparing DFAs with respect to their number of states is such a complexity ordering.

Definition 7. *A consistent learner that always constructs a smallest hypothesis with respect to a complexity ordering $\preceq$ on $\mathcal{H}$ is called an* $\preceq$-Occam learner.

Example 1. Consider $\mathcal{H} = \mathcal{C}$ to be the interval domain over the integers consisting of all intervals of the form $[l, r]$, where $l, r \in \mathbb{Z} \cup \{-\infty, \infty\}$ and $l \leq r$. We define $[l, r] \preceq [l', r']$ if either $[l, r] = [-\infty, \infty]$ or $\max\{|x| \mid x \in \{l, r\} \cap \mathbb{Z}\} \leq \max\{|x| \mid x \in \{l', r'\} \cap \mathbb{Z}\}$. For example, $[-4, \infty] \preceq [1, 7]$ because $4 \leq 7$. This ordering $\preceq$ satisfies the property that for each interval $[l, r]$ the set $\{[l', r'] \mid [l', r'] \preceq [l, r]\}$ is finite (because there are only finitely many intervals using integer constants with a bounded absolute value). A standard positive/negative sample $S = (P, N)$ with $P, N \subseteq \mathbb{N}$ is consistent with all intervals that contain the elements from P and do not contain an element from N. A learner that maps S to an interval that uses integers with the smallest possible absolute value (while being consistent with S) is an $\preceq$-Occam learner. For example, such a learner would map the sample $(P = \{-2, 5\}, N = \{-8\})$ to the interval $[-2, \infty]$. ⌟

The next theorem shows that $\preceq$-Occam learners ensure convergence in finite time (see the extended paper [31] for a proof).

Theorem 2. *If $\mathcal{T}$ is realizable and λ is a $\preceq$-Occam learner, then λ converges. Furthermore, the learner converges to a $\preceq$-minimal target element.*

There are several existing algorithms in the literature that use such orderings to ensure convergence. Several enumeration-based solvers are convergent because of the ordering of enumeration (e.g., the generic enumerative solver for SyGuS problems [2,1]). The invariant-generation ranging over conditional linear arithmetic expressions described in [16] ensures convergence using a total quasi-order based on the number of conditionals and the values of the coefficients. The learner uses templates to restrict the number of conditionals and a constraint-solver to find small coefficients for linear constraints.

Convergence using Tractable Well Founded Quasi-Orders The third strategy for convergence in finite time that we propose is based on well founded quasi-orders, or simply well-quasi-orders. Interestingly, we know of no existing learning algorithms in the literature that uses this recipe for convergence (a technique of similar flavor is used in [9]). We exhibit in this section a learning algorithm for intervals and for conjunctions of inequalities of numerical attributes based on this recipe. A salient feature of this recipe is that the convergence actually uses the samples returned by the teacher in order to converge (the first two recipes articulated above, on the other hand, would even guarantee convergence if the teacher just replies yes/no when asked whether the hypothesis is in the target set).

A binary relation $\preceq$ over some set X is a well-quasi-order if it is transitive and reflexive, and for each infinite sequence $x_0, x_1, x_2, \ldots$ there are indices $i < j$ such that $x_i \preceq x_j$. In other words, there are no infinite descending chains and no infinite anti-chains for $\preceq$.

Definition 8. *Let $(\mathcal{A}, \mathcal{T})$ be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$. A subset of hypotheses $\mathcal{W} \subseteq \mathcal{H}$ is called* wqo-tractable *if*
(a) there is a well-quasi-order $\preceq_{\mathcal{W}}$ on $\mathcal{W}$, and
(b) for each realizable sample $S \in \mathcal{S}$ with $\kappa_{\mathcal{H}}(S) \subseteq \mathcal{W}$, there is some $\preceq_{\mathcal{W}}$-maximal hypothesis in $\mathcal{W}$ that is consistent with S.

Example 2. Consider again the example of intervals over $\mathbb{Z} \cup \{-\infty, \infty\}$ with samples of the form $S = (P, N)$ (see Example 1). Let $p \in \mathbb{Z}$ be a point and let $\mathcal{I}_p$ be the set of all intervals that contain the point p. Then, $\mathcal{I}_p$ is wqo-tractable with the standard inclusion relation for intervals, defined by $[\ell, r] \subseteq [\ell', r']$ iff $\ell \geq \ell'$ and $r \leq r'$. Restricted to intervals that contain the point p, this is the product of two well-founded orders on the sets $\{x \in \mathbb{Z} \mid x \leq p\}$ and $\{x \in \mathbb{Z} \mid x \geq p\}$, and as such is itself well-founded [20, Theorem 2.3]. Furthermore, for each realizable sample (P, N), there is a unique maximal interval over $\mathbb{Z} \cup \{-\infty, \infty\}$ that contains P and excludes N. Hence, the two conditions of wqo-tractability are satisfied. (Note that this ordering on the set of *all* intervals is not a well-quasi-order; the sequence $[-\infty, 0], [-\infty, -1], [-\infty, -2], \ldots$ witnesses this.) ⌟

On a wqo-tractable $\mathcal{W} \subseteq \mathcal{H}$ a learner can ensure convergence by always proposing a maximal consistent hypothesis, as stated in the following lemma (a proof can be found in the extended paper [31]).

Lemma 2. *Let $\mathcal{T}$ be realizable, $\mathcal{W} \subseteq \mathcal{H}$ be wqo-tractable with well-quasi-order $\preceq_{\mathcal{W}}$, and S be a sample such that $\kappa_{\mathcal{H}}(S) \subseteq \mathcal{W}$. Then, there exists a learner that converges from the sample S.*

As shown in Example 2, for each $p \in \mathbb{Z}$, the set $\mathcal{I}_p$ of intervals containing p is wqo-tractable. Using this, we can build a convergent learner starting from the empty sample $\bot_s$. First, the learner starts by proposing the empty interval, the teacher must either confirm that this is a target or return a positive example, that is, a point p that is contained in every target interval. Hence, the set of hypotheses consistent with this sample is wqo-tractable and the learner can converge from here on as stated in Lemma 2. In general, the strategy for the learner is to force in one step a sample S such that the set $\kappa_{\mathcal{H}}(S) = \mathcal{I}_p$ is wqo-tractable. This is generalized in the following definition.

Definition 9. *We say that an* ALF is wqo-tractable *if there is a finite set $\{H_1, \ldots, H_n\}$ of hypotheses such that $\kappa_{\mathcal{H}}(S)$ is wqo-tractable for all samples S that are inconsistent with all H_i, that is, $\kappa_{\mathcal{H}}(S) \cap \{H_1, \ldots, H_n\} = \emptyset$.*

As explained above, the interval ALF is wqo-tractable with the set $\{H_1, \ldots, H_n\}$ consisting only of the empty interval.

Combining all the previous observations, we obtain convergence for wqo-tractable ALFs (see the extended paper [31] for a proof).

Theorem 3. *For every ALF instance $(\mathcal{A}, \mathcal{T})$ such that $\mathcal{A}$ is wqo-tractable and $\mathcal{T}$ is realizable, there is a convergent learner.*

A convergent learner for conjunctive linear inequality constraints. We have illustrated wqo-tractability for intervals in Example 2. We finish this section by showing that this generalizes to higher dimensions, that is, to the domain of n-dimensional hyperrectangles in $(\mathbb{Z} \cup \{-\infty, \infty\})^n$, which form the hypothesis space in this example. Each such hyperrectangle is a product of intervals over $(\mathbb{Z} \cup \{-\infty, \infty\})^n$. Note that hyperrectangles can, e.g., be used to model conjunctive linear inequality constraints over a set $f_1, \ldots, f_n : \mathbb{Z}^d \to \mathbb{Z}$ of numerical attributes.

The sample space depends on the type of target specification that we are interested in. We consider here the typical sample space of positive and negative samples (however, the reasoning below also works for other sample spaces, e.g., ICE sample spaces that additionally include implications). So, samples are of the form $S = (P, N)$, where P, N are sets of points in $\mathbb{Z}^n$ interpreted as positive and negative examples (as for intervals, see Example 1).

The following lemma provides the ingredients for building a convergent learner based on wqo-tractability (see the extended paper [31] for a proof).

Lemma 3. *(a) For each realizable sample $S = (P, N)$, there are maximal hyperrectangles that are consistent with S (possibly more than one).*
(b) For each $p \in \mathbb{Z}^n$, the set $\mathcal{R}_p$ of hyperrectangles containing p is well-quasi-ordered by inclusion.

We conclude that the following type of learner is convergent: for the empty sample, propose the empty hyperrectangle; for every non-empty sample S, propose a maximal hyperrectangle consistent with S.

4 Synthesis Problems Modeled as ALFs

In this section, we list a host of existing synthesis problems and algorithms that can be seen as ALFs. Specifically, we consider examples from the areas of program verification and program synthesis. We do not go into details of each formalism; instead, we refer the reader to the extended paper [31] for a thorough discussion. We encourage the reader to look up the referenced algorithms to better understand their mapping into our framework. Moreover, we have new techniques based on ALFs to compute fixed-points in the setting of abstract interpretation using learning; due to lack of space, we again refer the reader to the extended paper [31] regarding this.

Program Verification While program verification itself does not directly relate to synthesis, most program verification techniques require some form of help from the programmer before the analysis can be automated. Consequently, synthesizing objects that replace manual help has been an area of active research. We here focus on *learning loop invariants*. For the purposes of this article, let us consider while-programs with a single loop. Given a pre- and post-condition, assertions, and contracts for functions called, the problem is to find a loop invariant that proves the post-condition and assertions (assuming the program is correct).

A natural way to phrase this problem using ALFs is to model the concept space to consist of all subsets of program configurations and the hypothesis space

to be the set of all logical formulas capturing the class of invariants from which we want to synthesize. For a program, the target specification would be the set $\mathcal{T}_{Inv}$ of all inductive invariants that prove the post-condition and assertions correct.

A general approach to learning invariants, called *ICE learning* (for implication counterexamples), was recently proposed by Garg et al. [16], where the learner learns from positive, negative, and implication counterexamples. The corresponding ALF is $\mathcal{A}_{\text{ICE}} = (\mathcal{C}, \mathcal{H}, \gamma, \mathcal{S}, \kappa)$, where $\mathcal{C}$ is the set of all subsets of program configurations, the hypothesis space $\mathcal{H}$ is the language used to describe the invariant, and the sample space $\mathcal{S}$ is defined as follows:

- A sample is of the form $S = (P, N, I)$, where P, N are sets of program configurations (interpreted as positive and negative examples), and I is a set of pairs of program configurations (interpreted as implications).
- A set $C \in \mathcal{C}$ of program configurations is consistent with (P, N, I) if $P \subseteq C$, $N \cap C = \emptyset$, and if $(c, c') \in I$ and $c \in C$, then also $c' \in C$.
- The order on samples is defined by component-wise set inclusion; that is, $(P, N, I) \sqsubseteq_{\text{s}} (P', N', I')$ if $P \subseteq P'$, $N \subseteq N'$, and $I \subseteq I'$.
- The join is the component-wise union, and $\bot_{\text{s}} = (\emptyset, \emptyset, \emptyset)$.

Based on this ALF, we can now show that there always exists a teacher since a teacher can refute any hypothesis with a positive, a negative, or an implication counterexample, depending on which property of invariants is violated. Furthermore, we can show that having only positive and negative samples precludes the existence of teachers. In fact, we can show that if $\mathcal{C} = 2^D$ (for a domain D) and the sample space $\mathcal{S}$ consists of only positive and negative examples in D, then a target set $\mathcal{T}$ has a teacher *only if* it is defined in terms of excluding a set B and including a set G. Proofs can be found in the extended paper [31].

Several concrete implementations of ICE framework have been proposed: Garg et al.'s original work on ICE [16], the approach proposed by Sharma and Aiken [41] (which uses a learner based on stochastic search), the work by Garg et al. [15] on synthesizing quantified invariants for linear data structures such as lists and arrays, and the algorithm described by Neider [34].

Other non-ICE learning techniques for invariant synthesis that can be modeled as ALFs are the Houdini algorithm [14], which is implemented in the Boogie program verifier [7] and widely used (e.g., in verifying device drivers [29,30] and in race-detection in GPU kernels [8]) and parts of the learning-to-verify project [46], which leverages automata learning to the verification of infinite state systems.

Program Synthesis A prominent synthesis application is the sketch-based synthesis approach [42], where programmers write partial programs with holes and a system automatically synthesizes expressions or programs for these holes so that a specification (expressed using input-output pairs or logical assertions) is satisfied. The key idea is that given a sketch with a specification, we need expressions for the holes such that *for every possible input*, the specification holds. This roughly has the form $\exists \boldsymbol{e}. \forall \boldsymbol{x}. \psi(\boldsymbol{e}, \boldsymbol{x})$, where $\boldsymbol{e}$ are the expressions to synthesize and $\boldsymbol{x}$ are the inputs to the program.

The Sketch system implements a CEGIS technique using SAT solving, which works in rounds: the learner proposes hypothesis expressions and the teacher

checks whether $\forall \boldsymbol{x}.\, \psi(\boldsymbol{e}, \boldsymbol{x})$ holds (using SAT queries) and if not, returns a valuation for $\boldsymbol{x}$ as a counterexample. Subsequently, the learner asks, again using a SAT query, whether there exists a valuation for the bits encoding the expressions such that $\psi(\boldsymbol{e}, \boldsymbol{x})$ holds for every valuation of $\boldsymbol{x}$ returned by the teacher thus far; the resulting expressions are the hypotheses for the next round.

The above system can be modeled as an ALF. The concept space consists of tuples of functions modeling the various expressions to synthesize, the hypothesis space is the set of expressions (or their bit encodings), the map γ gives meaning to these expressions (or encodings), and the sample space can be seen as the set of *grounded formulae* of the form $\psi(\boldsymbol{e}, \boldsymbol{v})$ where the variables $\boldsymbol{x}$ have been substituted with a concrete valuation. The relation κ maps such a sample to the set of all expressions $\boldsymbol{f}$ such that the formulas in the sample all evaluate to true if $\boldsymbol{f}$ is substituted for $\boldsymbol{e}$. The Sketch learner can be seen as a learner in this ALF framework that uses calls to a SAT solver to find hypothesis expressions consistent with the sample.

The SyGuS format [2] is a competition format for synthesis, and extends the Sketch-based formalism above to SMT theories, with an emphasis on syntactic restrictions for expressions. More precisely, SyGuS specifications are parameterized over a background theory T, and an instance is a pair $(G, \psi(\boldsymbol{f}))$ where G is a grammar that imposes syntactic restrictions for functions (or expressions) and ψ is a formula in T, including function symbols $\boldsymbol{f}$; the functions $\boldsymbol{f}$ are typed according to domains of T. The goal is to find functions $\boldsymbol{g}$ for the symbols $\boldsymbol{f}$ in the syntax G such that ψ holds. SyGuS further restricts ψ to be of the form $\forall \boldsymbol{x}.\, \psi'(\boldsymbol{f}, \boldsymbol{x})$ where ψ' is a quantifier-free formula in a decidable SMT theory.

There have been several solvers developed for SyGuS (cf. the first SyGuS competition [2,1]), and all of them are in fact learning-based (i.e., CEGIS). In particular, three solvers have been proposed: an enumerative solver, a constraint-based solver, and a stochastic solver. All these solvers can be seen as ALF instances: the concept space consists of all tuples of functions over the appropriate domains and the hypothesis space is the set of all functions allowed by the *syntax* of the problem (with the natural γ relation giving its semantics). Note that the learners *know* ψ in this scenario. However, we can model SyGuS as ALFs by taking the sample space to be grounded formulas $\psi'(\boldsymbol{f}, \boldsymbol{v})$ consisting of the specification with particular values $\boldsymbol{v}$ substituted for $\boldsymbol{x}$. The learners can now be seen as learning from these samples, without knowledge of ψ (similar to Sketch above).

We would like to emphasize that this embedding of SyGuS as an ALF clearly showcases the difference between different synthesis approaches (as mentioned in the introduction). For example, invariant generation can be done using learning either by means of ICE samples or modeled as a SyGuS problem. However, it turns out that the sample spaces (and, hence, the learners) in the two approaches are *very different*! In ICE-based learning, samples are only single configurations (labeled positive or negative) or pairs of configurations, while in a SyGuS encoding, the samples are grounded formulas that encode the entire program body, including instantiations of universally quantified variables at intermediate states in the execution of the loop.

Similarly, for synthesizing linear arithmetic expressions, there are again different kinds of solvers. The SyGuS solvers are based on the sample space of grounded formulae as above, while certain other solvers of the Alchemist variety [39,35] are based on a different sample space that involve counterexamples that encode inputs on which the hypothesis is incorrect; these two classes are consequently very different from each other (for instance, the latter use machine-learning techniques to classify inputs that cannot be achieved with the former kind of sample).

There are several algorithms that are self-described as CEGIS frameworks, and, hence, can be modeled using ALFs. For example, synthesizing loop-free programs [19], synthesizing synchronizing code for concurrent programs [10] (in this work, the sample space consists of abstract concurrent partially-ordered traces), work on using synthesis to *mine specifications* [22], synthesizing bit-manipulating programs and deobfuscating programs [21] (here, the use of separate I/O-oracle can be modeled as the teacher returning the output of the program together with a counterexample input), superoptimization [40], deductive program repair [26], synthesis of recursive functional programs over unbounded domains [27], as well as synthesis of protocols using enumerative CEGIS techniques [45]. Finally, an example for employing a human as teacher is FLASHFILL by Gulwani et al. [18], which synthesizes string manipulation macros from user-given input-output examples in the context of Microsoft Excel.

5 Conclusions

We have presented an abstract learning framework for synthesis that encompasses several existing techniques that use learning or counter-example guided inductive synthesis to create objects that satisfy a specification. (We refer to the extended paper [31] for a discussion of extensions and limitations of our abstract learning framework.) We were motivated by abstract interpretation [12] and how it gives a general framework and notation for verification; our formalism is an attempt at such a generalization for learning-based synthesis. The conditions we have proposed that the abstract concept spaces, hypotheses spaces, and sample spaces need to satisfy to define a learning-based synthesis domain seem to be cogent and general in forming a vocabulary for such approaches. We have also addressed various strategies for convergent synthesis that generalizes and extends existing techniques (again, in a similar vein as to how widening and narrowing in abstract interpretation give recipes for building convergent algorithms to compute fixed-points). We believe that the notation and general theorems herein will bring more clarity, understanding, and reuse of learners in synthesis algorithms.

Acknowledgements: This work was partially supported by NSF Expeditions in Computing ExCAPE Award #1138994.

References

1. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Comm. Security, vol. 40, pp. 1–25. IOS Press (2015)

2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD 2013. pp. 1–8. IEEE (2013)
3. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL 2005. pp. 98–109. ACM (2005)
4. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: Results and analysis of sygus-comp'15. Tech. rep., University of Pennsylvania (2015), http://www.cis.upenn.edu/~fisman/documents/AFSS_SYNT15.pdf
5. Angluin, D.: Computational learning theory: Survey and selected bibliography. In: STOC 1992. pp. 351–369. ACM (1992)
6. Baker, A.: Simplicity. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Fall 2013 edn. (2013), http://plato.stanford.edu/archives/fall2013/entries/simplicity/
7. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. pp. 364–387 (2005)
8. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: Gpuverify: a verifier for GPU kernels. In: OOPSLA 2012. pp. 113–132. ACM (2012)
9. Blum, A.: Learning boolean functions in an infinite attribute space. Machine Learning 9, 373–386 (1992)
10. Cerný, P., Clarke, E.M., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Samanta, R., Tarrach, T.: From non-preemptive to preemptive scheduling using synchronization synthesis. In: CAV 2015. LNCS, vol. 9207, pp. 180–197. Springer (2015)
11. Cheung, A., Madden, S., Solar-Lezama, A., Arden, O., Myers, A.C.: Using program analysis to improve database applications. IEEE Data Eng. Bull. 37(1), 48–59 (2014)
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
13. Domingos, P.M.: The role of occam's razor in knowledge discovery. Data Min. Knowl. Discov. 3(4), 409–425 (1999)
14. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)
15. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV'13. vol. 8559, pp. 813–829. Springer (2013)
16. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV 2014. vol. 8559, pp. 69–87. Springer (2014)
17. Garg, P., Madhusudan, P., Neider, D., Roth, D.: Learning invariants using decision trees and implication counterexamples. POPL 2016 p. to appear (2016)
18. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL 2011. pp. 317–330. ACM (2011)
19. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI 2011. pp. 62–73. ACM (2011)
20. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proc. London Math. Soc. s3-2(1), 326–336 (1952)
21. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE 2010. pp. 215–224. ACM (2010)
22. Jin, X., Donzé, A., Deshmukh, J.V., Seshia, S.A.: Mining requirements from closed-loop control models. In: HSCC 2013. pp. 43–52. ACM (2013)

23. Karaivanov, S., Raychev, V., Vechev, M.T.: Phrase-based statistical translation of programming languages. In: Onward!, SLASH '14. pp. 173–184. ACM (2014)
24. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press, Cambridge, MA, USA (1994)
25. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: AAIP 2009, Revised Papers. Lecture Notes in Computer Science, vol. 5812, pp. 50–73. Springer (2010)
26. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: CAV 2015. LNCS, vol. 9207, pp. 217–233. Springer (2015)
27. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA 2013. pp. 407–426. ACM (2013)
28. Kuncak, V.: Verifying and synthesizing software with recursive functions - (invited contribution). In: ICALP 2014. LNCS, vol. 8572, pp. 11–25. Springer (2014)
29. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: (FSE-22). pp. 202–212. ACM (2014)
30. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer (2012)
31. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. Tech. rep., University of Illinois at Urbana-Champaign (2016), `http://madhu.cs.illinois.edu/tacas16b/`
32. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. 2(1), 90–121 (Jan 1980)
33. Mitchell, T.M.: Machine learning. McGraw-Hill (1997)
34. Neider, D.: Applications of Automata Learning in Verification and Synthesis. Ph.D. thesis, RWTH Aachen University (April 2014)
35. Neider, D., Saha, S., Madhusudan, P.: Synthesizing piece-wise functions by learning classifiers. In: TACAS 2016. p. to appear. LNCS, Springer (2016)
36. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: PLDI'15. pp. 619–630. ACM (2015)
37. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL'89. pp. 179–190 (1989)
38. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
39. Saha, S., Garg, P., Madhusudan, P.: Alchemist: Learning guarded affine functions. In: CAV 2015. LNCS, vol. 9206, pp. 440–446. Springer (2015)
40. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS '13. pp. 305–316. ACM (2013)
41. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer (2014)
42. Solar-Lezama, A.: Program Synthesis by Sketching". Ph.D. thesis, University of California at Berkeley (2008)
43. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415 (2006)
44. Thakur, A., Lal, A., Lim, J., Reps, T.: PostHat and all that: Attaining most-precise inductive invariants. Tech. Rep. TR1790, University of Wisconsin, Madison (Apr 2013)
45. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: PLDI'13. pp. 287–296. ACM (2013)
46. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: ICFEM 2004. vol. 3308, pp. 274–289. Springer (2004)