Abstract Learning Frameworks for Synthesis

Christof Löding
RWTH Aachen University, Germany
loeding@cs.rwth-aachen.de

P. Madhusudan
University of Illinois at Urbana-Champaign, USA
madhu@illinois.edu

Daniel Neider
University of Illinois at Urbana-Champaign, USA
neider2@illinois.edu

Abstract

We develop abstract learning frameworks (ALFs) for synthesis that embody the principles of CEGIS (counter-example based inductive synthesis) strategies that have become widely applicable in recent years. Our framework defines a general abstract framework of iterative learning, based on a hypothesis space that captures the synthesized objects, a sample space that forms the space on which induction is performed, and a concept space that abstractly defines the semantics of the learning process. We show that a variety of synthesis algorithms in current literature can be embedded in this general framework. While studying these embeddings, we also generalize some of the synthesis problems these instances are of, resulting in new ways of looking at synthesis problems using learning. We also investigate convergence issues for the general framework, and exhibit three recipes for convergence in finite time. The first two recipes generalize current techniques for convergence used by existing synthesis engines. The third technique is a more involved technique of which we know of no existing instantiation, and we instantiate it to concrete synthesis problems.

1. Introduction

The field of synthesis, which includes several forms of synthesis including synthesizing controllers, program expressions, program repairs, program translations, system invariants, ranking functions, and even entire programs, has become a fundamental and vibrant subfield in programming languages. While classical studies of synthesis had focused on synthesizing entire programs or controllers from specifications [35, 40], 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 (finding program expressions that complete code), synthesizing Excel programs for string transformations [22], synthesizing superoptimized code [43], deobfuscating code [26], synthesizing invariants to help in verification [20, 21], etc.

One prominent technique that has emerged in recent years for expression synthesis is based on *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 iterative 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 *iterative learning-based approach to synthesis* is very similar to the counter-example guided inductive synthesis (CEGIS) approach [45] in current literature, which philosophically advocates precisely this kind of inductive synthesis. The iterative learning-based synthesis 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 [20, 21] to specification mining [1], program expressions that complete sketches [46], superoptimization [43], control [27], string transformers for spreadsheets [22], protocols [49], etc. Specifications for synthesis tend to be extremely varied and complex— ranging from complex programs or complex universally quantified formulae to simple but underspecified requirements like a few examples. Consequently, the learning-based approach which divorces itself from looking at the specification but looks only on the hypotheses space of objects to synthesize and simple conditions imposed by facets of these specifications has turned out to be very effective. Moreover, when the specification is underspecified, the synthesis engine effectively *generalizes* the specification, using the natural learning biases in the learning algorithms.

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 learning-based synthesis across different domains.

An abstract learning framework (ALF) (see Figure 1 on Page 3) 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 then 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, and hence the modeling of the target space is very important. In synthesis problems, the teacher does not have one target in mind nor does she know explicitly the target set (if she knew this, there would be no reason to use learning as we can just take some element of the target space). 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 program/loop, and this defines implicitly a *set* of invariants as target; in program deobfuscation, she knows that the synthesized program should be equivalent to the given program. Furthermore, a teacher can be built when we can have an algorithm that can examine a hypothesis and check whether it satisfies the properties that define 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.

The learner and teacher (verification oracle) are then couched in this abstract learning framework: the learner maps samples to hypotheses while the teacher proposes new samples that exhibit why the proposed hypothesis is not in the target set. The precise notions of what it means for a hypothesis to be consistent with a sample, and what it means for a teacher to return a sample that distinguishes the hypothesis from the target are modeled abstractly in our framework. The learner and teacher work in tandem forever, or until the learner synthesizes a concept in the 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 [46], synthesizing loop-free programs [23], mining specifications [27], synthesizing synchronizations code for concurrent programs, etc.), (b) synthesis from input-output examples such as Flashfill [22], (c) the CEGIS framework applied to the concrete problem of solving synthesis problems expressed in the SMT-based SyGuS format [2, 3], and three synthesis engines that use learning to synthesize solutions, (d) invariant synthesis frameworks, including Houdini [17] and the more recent ICE-learning model for synthesizing loop invariants [20], spanning a variety of domains from arithmetic [20, 21] to quantified invariants over data structures [19], and (e) synthesizing fixed-points and abstract transformers in abstract interpretation settings [47].

The import of embedding various existing synthesis algorithms in our framework is that it helps clarify the precise learning techniques used, spelling out the sample space, the meaning of samples, and the learning algorithms that work using these samples. Nuances of multiple learning-based synthesis algorithms for the *same* problem can be differentiated more clearly when formalized as ALFs. One such insight that we obtained during this work concerns invariant synthesis. A recent scheme for invariant synthesis uses the ICE learning model, where learning-based synthesis synthesizes an invariant using samples that are either single program configurations labeled positive/negative or *implication counter-examples* consisting of pairs of program configurations [20]. Invariant synthesis (for programs with linear arithmetic bodies and invariants) can also be formulated in the general SyGuS specification formulation that asks to synthesize an expression e that satisfies $\forall \vec{x}.\ \psi(e, \vec{x})$, where ψ itself is in a quantifier-free decidable theory. Hence, one would expect that CEGIS-solvers for SyGuS would solve the problem in the same way as ICE does; however, this is far from true! The sample space for ICE-learners are single or pairs of program configurations, while the SyGuS learner *knows* ψ, which includes the semantics of the program being encoded, etc. as well as a valuation of intermediate variables used in the body of the program! Consequently, the solvers work very differently.

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. Furthermore, the learning algorithms are accurately described by the abstract framework, and hence allows us to *swap learners* while keeping the rest of the context the same. For example, when hypotheses spaces consist of conjunctions of predicates, and samples consist of valuations of predicates, the learning problem is a *conjunctive Boolean formula* learning problem, and there are several efficient machine learning algorithms for this problem [12, 28, 34, 36]. Applications that use such a sample domain (like Houdini [17] in program verification and Daikon [16] for software property mining) can utilize these learners interchangeably. Similarly, for problems in constructing deobfuscation of functions from integers to integers [26], we can use guarded linear arithmetic learning algorithms like [42], when the sample spaces and hypothesis spaces match (see Section 4).

Convergence

Our second main contribution 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 sets), 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 properties of 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, then 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 [17], etc., learning Boolean functions (like decision-tree learning with

purely Boolean predicates as attributes) or functions over bit-vector domains (Sketch [46] 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 [19].

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 [28] and [36]) though its universal appeal in generalizing concepts is debatable [15]. 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 most of the convergent learning mechanisms that we know of in the literature (for example, the convergent ICE-learning algorithms for synthesizing invariants using constraint solvers [20], the enumerative solvers in almost every domain of synthesis [30, 31, 38, 49], including for SyGuS [2, 3], that enumerate by dovetailing through expressions).

The third recipe for finite convergence is a more complex one based on well-founded quasi orderings and the existence of maximal hypotheses with respect to this ordering for any set of samples that results after a bounded set of hypotheses by the learner. 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. We 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 a domain of integers.

The 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 paper is structured as follows. We present the theory of abstract learning frameworks for synthesis in Section 2 and the various general recipes for convergence in Section 3 (including the two new learning algorithms following the third recipe), explaining the theory using simple synthesis domains. In Section 4 we exhibit how a variety of learning-based synthesis algorithms in the literature can be formulated using our framework, and show the variety of algorithms our framework can capture and contrast some of the nuances of the algorithms that come about with the formulations. While the goal of this section is mainly to formulate existing synthesis engines in the abstract framework, we also present some new applications of our framework, in particular a generalization of ICE-learning to learning fixed-points in abstract interpretation. Section 5 discusses some variations and limitations of our framework, and Section 6 concludes.

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).

An abstract learning framework (ALF) consists of a concept space $\mathcal{C}$, a hypothesis space $\mathcal{H}$ and a sample space

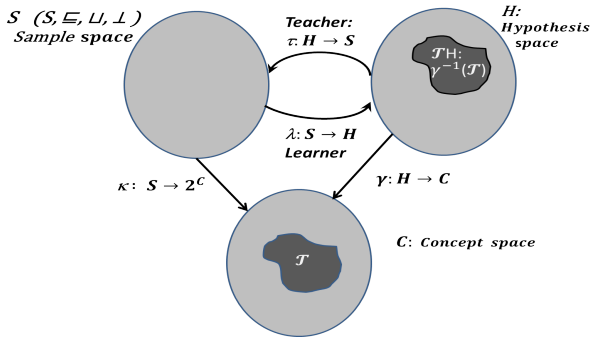

Figure 1. Overview of the components of an abstract learning framework

$\mathcal{S}$. The hypothesis space represents the set of concepts that the learner learns using samples from the sample space $\mathcal{S}$. The *semantics* relating the samples to hypothesis is given using the concept space $\mathcal{C}$, and appropriate maps between them (γ and κ).

An *abstract learning framework for synthesis* (ALF, for short), will consist of a tuple of the form $\mathcal{A} = (\mathcal{C}, \mathcal{H}, \gamma, \mathcal{S}, \kappa)$, where $\mathcal{C}$ is a concept space, $\mathcal{H}$ is a hypothesis space, $\gamma : \mathcal{H} \to \mathcal{C}$ is a concretization function, $\mathcal{S}$ is a sample space, and $\kappa : \mathcal{S} \to 2^{\mathcal{C}}$ is a consistency function.

We explain these components in more detail in the following. Some of these components have restrictions which we will motivate before defining ALFs formally.

As in computational learning theory, as presented in e.g., in [5] or [28], we consider a *concept space* $\mathcal{C}$, which contains the objects that we are interested in. For example, in a verification setting, 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, like all functions from $\mathbb{Z}^n$ to $\mathbb{Z}$.

The *hypothesis space* $\mathcal{H}$ contains the objects that the learner produces. These are representations of (some) elements from the concept space; prime examples of hypotheses spaces are logical terms or formulas in a syntactically restricted logic. 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). For the concept and hypothesis space in the example above, this function maps each syntactic function to the semantic function corresponding to it.

In classical computational learning theory for classification [28, 36], one often considers samples consisting of positive and negative examples. These samples are supposed to provide information on a target concept that is to be learned. 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).

The space of samples used for learning is therefore not standard positive/negative examples in our framework, but an abstract space . 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 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* captures the semantics of samples with respect to the concept space. Formally, it is given as a function $\kappa : \mathcal{S} \to 2^{\mathcal{C}}$ that assigns to each sample S the set $\kappa(S)$ of concepts that are consistent with the sample.

We require the consistency relation to satisfy $\kappa(\bot_s) = \mathcal{C}$, and $\kappa(S_1 \sqcup S_n) = \kappa(S_1) \cap \kappa(S_2)$ for all $S_1, S_2 \in \mathcal{S}$. The first condition 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. [1] The second property implies monotonicity of the consistency relation in the following sense.

Remark 2.1. *If $S_1 \sqsubseteq_s S_2$, then $\kappa(S_2) \subseteq \kappa(S_1)$.*

We say that the sample space is complete if $(\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s)$ is a complete bounded join-semilattice, i.e., if the join is defined for arbitrary subsets of $\mathcal{S}$. In this case, the consistency relation has to satisfy the following property for each $\mathcal{S}' \subseteq \mathcal{S}$:

$$\bigcap_{S \in \mathcal{S}'} \kappa(S) = \kappa(\bigsqcup(\mathcal{S}')).$$

We can now define the framework formally:

Definition 2.2 (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: $\kappa : \mathcal{S} \to 2^{\mathcal{C}}$*

and where the following conditions hold:

- $\kappa(\bot_s) = \mathcal{C}$
- $\kappa(S_1 \sqcup S_n) = \kappa(S_1) \cap \kappa(S_2)$ *for all* $S_1, S_s \in \mathcal{S}$.

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. □

Some other auxiliary definitions we will need: We define $\kappa_{\mathcal{H}}(S) := \gamma^{-1}(\kappa(S))$ 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 is a hypothesis that is consistent with S, that is, $\kappa_{\mathcal{H}}(S) \neq \emptyset$.

[1] Aside: The condition that $\kappa(S_1 \sqcup S_n) \subseteq \kappa(S_1) \cap \kappa(S_2)$ 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 $\kappa(S_1 \sqcup S_n) \supseteq \kappa(S_1) \cap \kappa(S_2)$ is debatable; it claims that any concept consistent with both S_1 and S_2 must be consistent with their join, or in other words, that samples when taken together cannot eliminate a concept that they couldn't eliminate individually. However, we have found no natural framework that requires such a generalization, and this condition does simplify the framework significantly because it ensures that the property of honesty of teachers holds in an iterative setting (see Definition 2.5 and Lemma 2.7 below).

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 2.3 (ALF Instance)**.** *An ALF instance is a pair $(A, \mathcal{T})$ where $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$ 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 $\mathcal{T}$*. (If she did, there'd be no point in building a teacher and a learner, as we can just return some element of the target set. In fact, she does not even know whether the target set is empty.) Rather she knows only certain properties that define the target space. And when presented with a hypothesis concept H by the learner, the teacher can examine whether H satisfies the *properties required of the target set*, and give samples, based on the failed properties, that show why H is not in 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, though it does not explicitly know the invariants of the program.

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 hypothesis spaces, and a consistent learner to be a learner that only proposes consistent hypotheses for samples.

Definition 2.4. *A learner for an ALF $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$ 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. $\bot_s$). 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. This property is referred to as *progress* because the learner obtains new information that rules out the current hypothesis. Furthermore, we require that the teacher is *honest* in the sense that it does not provide feedback that rules out target elements.

Definition 2.5. *Let $(A, \mathcal{T})$ be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$, 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: $\tau(H) = \bot_s$ *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* $H \in \mathcal{H}$. □

The first condition, progress, says that if the hypothesis is in the target set, then the teacher must return the "empty" sample $\bot_s$, signaling that the learner has learned a target. Otherwise, the teacher must return a sample that rules out the current hypothesis (i.e., the current hypothesis should not be consistent with the returned sample). This ensures that a consistent learner can never propose the same hypothesis again, and hence makes progress.

The second condition, 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. If we iterate this interaction, starting from the empty sample, we obtain an infinite sequence of samples. In case of a complete sample lattice, this is a sequence indexed by ordinals. If the sample is not complete, we obtain a sequence indexed by natural numbers. The iterative interaction between the learner and teacher is formally defined as follows:

Definition 2.6. *Let* $(A, \mathcal{T})$ *be an ALF instance, with* $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \bot_s), \gamma, \kappa)$, *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.*

Then 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, by iterative application of* $f_{\tau,\lambda}$*:*

- $S^0_{\tau,\lambda} := \bot_s$
- $S^{\alpha+1}_{\tau,\lambda} := f_{\tau,\lambda}(S^\alpha_{\tau,\lambda})$ *for successor ordinals*
- $S^\alpha_{\tau,\lambda} := \bigsqcup_{\beta<\alpha} S^\beta_{\tau,\lambda}$ *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. The first property below says that the set of concepts consistent with the hypotheses along the sequence of interaction strictly decreases. The second property says that no element of the target set is ever inconsistent with any hypothesis in the sequence.

Lemma 2.7. *If* $\mathcal{T}$ *is realizable,* $\mathcal{S}$ *is a complete sample lattice,* λ *is a consistent learner, and* τ *is a teacher, 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* $\lambda(S^\alpha_{\tau,\lambda}) \in \mathcal{T}$,

(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* $\alpha \in \mathbb{N}$.

Proof. The proof is a straight-forward transfinite induction, using the properties of the teacher and the consistency relation. For the case of non-complete sample lattice, ignore the limit step in the proof below.

For part (b), the induction base is given by $\kappa(\bot_s) = \mathcal{C}$. The induction step for limit ordinals directly follows from the property of the consistency relation: all previous samples are consistent with the target specification, so their join is, too. For a successor ordinal $\alpha + 1$ it follows from the fact that $S^{\alpha+1}_{\tau,\lambda}$ is a join of two samples that are both consistent with the target specification.

For part (a), let $\alpha \in \mathbb{O}$ such that $H := \lambda(S^\alpha_{\tau,\lambda}) \notin \mathcal{T}$. Then $S^{\alpha+1}_{\tau,\lambda} = S^\alpha_{\tau,\lambda} \cup S$ with $S = \tau(H)$. In particular, $S^\alpha_{\tau,\lambda} \sqsubseteq_s S^{\alpha+1}_{\tau,\lambda}$ and thus $\kappa(S^\alpha_{\tau,\lambda}) \supseteq \kappa(S^{\alpha+1}_{\tau,\lambda})$ by Remark 2.1. For the strictness of the inclusion, note that $\gamma(H) \in \kappa(S^\alpha_{\tau,\lambda})$ because λ is a consistent learner (and $S^\alpha_{\tau,\lambda}$ is realizable because it is consistent with the realizable target specification by (b)). Furthermore, $\gamma(H) \notin \kappa(S)$ by the progress property of the teacher, and hence $\gamma(H) \notin \kappa(S^{\alpha+1}_{\tau,\lambda})$. □

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 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 $(\vec{i}, o$, which is a concrete input $\vec{i}$ on which h differs from P, and o is the output of P on $\vec{i}$. Then the sample space would consist of sets of such pairs (with union for join and empty set for $\bot_s$). The iterative learning will then model the process of synthesis using learning of 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.

3.1 Transfinite convergence

From Lemma 2.7 one can conclude that the transfinite sequence of hypotheses constructed by the learner converges to a target set.

Theorem 3.1. *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* $\lambda(S^\alpha_{\tau,\lambda}) \in \mathcal{T}$.

Proof. Let α be an ordinal with cardinality bigger than $|\mathcal{H}|$ (bigger than $|\mathcal{S}|$ also works). If $\lambda(S^\beta_{\tau,\lambda}) \notin \mathcal{T}$ for all $\beta < \alpha$, then Lemma 2.7 (a) implies that all $\lambda(S^\beta_{\tau,\lambda})$ for $\beta < \alpha$ are pairwise different, which contradicts the cardinality assumption. □

The above theorem ratifies the choice of our definitions, and the proof (relying on Lemma 2.7) 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.).

3.2 Convergence in finite time

Convergence in finite time is clearly the most desirable notion, and we propose tactics for designing learners that convergence 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 $S \neq \bot_s$ (that is, if $S^0_{\lambda,\tau} = S$).

3.2.1 Finite hypotheses spaces

We first note that if the hypothesis space $\mathcal{H}$ is finite, then convergence is guaranteed whenever we have a consistent learner. By Lemma 2.7, the learner always makes progress, and hence cannot propose the same hypothesis twice. Consequently, the learner can only produce a finite number of hypotheses before finding one that's in the target (or declare that no such hypothesis exists).

There are several synthesis engines using learning that use finite hypotheses spaces. For example, Houdini [17] is a learner of *conjunctions* over a fixed finite set of predicates, and hence has a finite hypotheses space. Learning decision trees over purely Boolean attributes (not numerical) [41] is also convergent because of finite hypotheses spaces, and this extends to the ICE learning model as well [21]. Invariant generation for arrays and lists using *elastic QDAs* [19] also uses a convergent argument that relies on a finite hypothesis space.

3.2.2 Occam Learners

We now show the most robust strategy we know for convergence, based on building learners that follow the Occam razor. 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 (for example, *pruning* in decision-tree learning [36]), though the general applicability of Occam's razor in machine learning as a sound means to generalize is debatable [15]. In this section, we 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. Such a learner, in an iterative learning based synthesis setting will always converge 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, even if $H \neq H'$.

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

Definition 3.2. *A* complexity ordering *is a total quasi-order* $\preceq$ *such that for each* $x \in \mathcal{H}$ *the set* $\{y \in \mathcal{H} \mid y \leq x\}$ *is finite.*

The example of comparing DFAs w.r.t. their number of states is such a complexity ordering.

Definition 3.3. *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 3.4. *Consider* $\mathcal{H}$ *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 the maximal absolute value of the integers appearing in* $\{l, r\}$ *is at most as big as the maximal absolute value of the integers appearing in* $\{l', r'\}$*. For example,* $[-4, \infty] \preceq [1, 7]$ *because* $4 \leq 7$*. Furthermore,* $[-\infty, \infty]$ *is smaller than all other intervals w.r.t.* $\preceq$*.*

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 the are only finitely many intervals using constants with a bounded absolute value).*

For this example, the hypothesis space and the concept space are the same. As samples we use pairs $S = (P, N)$*, where* $P, N \subseteq \mathbb{N}$*. Such a sample is consistent with all intervals that contain the elements from* P *and does not contain the elements from* N*.*

Let λ *be learner that maps* S *to an interval that uses integers with the smallest possible absolute value (while being consistent with* S*). Then* λ *is an* $\preceq$*-Occam learner. For example, such a learner would map the sample* $(\{-2, 5\}, \{-8\})$ *to the interval* $[-2, \infty]$*.*

The next theorem shows that $\preceq$-Occam learners ensure convergence in finite time.

Theorem 3.5. *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.*

Proof. Pick any target element $T \in \mathcal{H}$, which exists because $\mathcal{T}$ is realizable. Since τ is honest, $T \in \kappa(S^n_{\tau,\lambda})$ for all n by Lemma 2.7(b). Thus, on the iterated sample sequence, a $\preceq$-Occam learner never constructs an element which is strictly above T w.r.t. $\preceq$. Since there are only finitely many hypothesis that are not strictly above T, and since the learner always makes progress according to Lemma 2.7, it converges to a target element in finitely many steps, which itself does not have any other target elements below, and thus is $\preceq$-minimal. □

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 (for example, the generic enumerative solver for SyGus problems [2, 3]. Another example in the work reported in [20], invariant-generation ranging over conditional linear arithmetic expressions can be biased towards those that have smaller number of conditionals and smaller coefficients. This defines a total quasi-order, and the learner uses *templates* to restrict the number of conditionals and a constraint-solver to find coefficients for linear constraints and constrains them further with the simplicity measure, thereby assuring convergence.

3.2.3 Convergence using Tractable Well Founded Quasi-Orders

The third strategy for convergence in finite time 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. 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 3.6. *Let $(\mathcal{A}, \mathcal{T})$ be an ALF instance, where $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$.*
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 3.7. *Consider again the example of intervals over $\mathbb{Z} \cup \{-\infty, \infty\}$ with samples of the form $S = (P, N)$ (see Example 3.4). Let $p \in \mathbb{N}$ be some 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 as follows: $[\ell, r] \subseteq [\ell', r']$ 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 $\{x \in \mathbb{N} \mid x \leq p\}$ and $\{x \in \mathbb{N} \mid x \geq p\}$, and as such is itself well-founded [24, 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 set of all *intervals is not a well-quasi-order; the sequence $[-\infty, 0], [-\infty, -1], [-\infty, -2], \ldots$ witnesses this.)*

Lemma 3.8. *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 is a learner that converges from the sample S.*

Proof. For any sample S' with $S \sqsubseteq_{\mathrm{s}} S'$, the set $\kappa_{\mathcal{H}}(S')$ of hypotheses consistent with S' is a subset of $\mathcal{W}$. Therefore, there is some $\preceq_{\mathcal{W}}$-maximal element in $\mathcal{W}$ that is consistent with S'. The strategy of the learner is to return such a maximal hypothesis. Assume, for the sake of contradiction, that such a learner does not converge from S for some teacher τ. Let $H_0, H_1, \ldots$ be the infinite sequence of hypothesis produced by λ and τ starting from S, and let $S_0, S_1, S_2, \ldots$ be the corresponding sequence of samples (with $S_0 = S$). The well-foundedness of $\preceq_{\mathcal{W}}$ implies that there are $i < j$ with $H_i \preceq_{\mathcal{W}} H_j$. However, $S_i \sqsubseteq_{\mathrm{s}} S_j$ because S_j is obtained from S_i by joining answers of the teacher. Therefore, H_j is also consistent with S_i (Remark 2.1). This contradicts the choice of H_i as a maximal hypothesis that is consistent with S_i. □

As shown in Example 3.7, 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 $\bot_{\mathrm{s}}$. 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 using the strategy described in the proof above.

Hence 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 3.9. *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.

Theorem 3.10. *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.*

Proof. A convergent learner can be built as follows. Let $\{H_1, \ldots, H_n\}$ be the finite set of hypotheses from the definition of wqo-tractability of an ALF.

- As long as the current sample is consistent with some H_i, propose such an H_i.
- Otherwise, the current sample S is such that $\kappa_{\mathcal{H}}(S)$ is wqo-tractable, and thus the learner can apply the strategy from Lemma 3.8. □

Note that Theorem 3.10 only requires $\mathcal{T}$ to be realizable. The property of being wqo-tractable is a property of $\mathcal{A}$, independent of $\mathcal{T}$. This means, that the learning strategy from the proof of Theorem 3.10 is convergent for every target specification.

A convergent learner for conjunctive linear inequality constraints. We have illustrated wqo-tractability for intervals in Example 3.7. We finish this section by showing that this generalizes to higher dimensions, that is, to the domain of n-dimensional hyperrectangles in $(\mathbb{Z} \cup \{-\infty, \infty\})^n$. Each such hyperrectangle is a product of intervals over $(\mathbb{Z} \cup \{-\infty, \infty\})^n$.

Note that hyperrectangles can, for example, be used to model conjunctive linear inequality constraints over a set of numerical attributes over a domain of integers. More precisely, assume that the concept space consists of sets over $\mathbb{Z}^d$, and the target specification is described in terms of functions $f_1, \ldots, f_n : \mathbb{Z}^d \to \mathbb{Z}$.

Now hyperrectangles over $(\mathbb{Z} \cup \{-\infty, \infty\})^n$ capture conjunctions of constraints of the form $f_i \leq m$ or $f_i \geq m$, where $m \in \mathbb{Z}$. The concretization function maps a conjunction of constraints to the set of all points in $\mathbb{Z}^d$ that satisfy all the constraints from the conjunction ($+/-\infty$ is used in dimension i if f_i is not constrained from above or below). Therefore, building convergent learners for hyperrectangles yields convergent learners for an interesting class of numerical properties.

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 the 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 3.4.

For each realizable sample $S = (P, N)$, there are maximal hyperrectangles that are consistent with S (possibly more than one), because for each increasing chain $R_0 \subseteq R_1 \subseteq \cdots$ of hyperrectangles that are all consistent with S, the union $\bigcup_{i \geq 0} R_i$ is also a hyperrectangle that is consistent with S.

Now consider the following type of learner:

- For the empty sample, propose the empty hyperrectangle.
- For every non-empty sample S, propose a maximal hyperrectangle consistent with S.

We claim that this learner converges. In the first round, the teacher returns a positive example, that is, a point $p \in \mathbb{Z}^n$ that is contained in every target hyperrectangle (we assume here that a teacher exists, which means in particular that it can return a sample for the empty hyperrectangle). We now show that the set $\mathcal{R}_p$ of hyperrectangles containing a point p is well-quasi-ordered by inclusion. Then the proposed learner is of the form as described in the proof of Theorem 3.10, and thus convergent.

For a point $p = (p_1, \ldots, p_n)$, the set $\mathcal{R}_p$ is the product $\mathcal{R}_p = \mathcal{I}_{p_1} \times \cdots \times \mathcal{I}_{p_n}$ of the sets of intervals containing the points p_i. Furthermore, the inclusion order for hyperrectangles is the n-fold product of the inclusion order for intervals. Thus, the inclusion order on $\mathcal{R}_p$ is a well-quasi-order because it is a product of well-quasi-orders [24].

4. Synthesis Problems Modeled as ALFs

The goal of this section is to describe how a host of existing synthesis problems and algorithms can be viewed as ALFs (abstract learning frameworks), describing the concept, hypotheses, and sample spaces, and showing how the learner works with a teacher to synthesize the required objects in each domain. We cannot go into each such synthesis algorithm in detail nor establish the map formally; we encourage the reader to look up the referenced algorithms in order to better understand their mapping into our framework. The embedding into our framework also showcases the variety of hypotheses and sample classes that exist today, and in some cases, shows the advantages our vocabulary offers in capturing the nuances and differences, especially in different synthesis solutions to the same problem.

4.1 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 such manual help has been an area of active research. We focus on two such objects: *learning loop invariants* and *abstract transformers*.

4.1.1 Invariant Synthesis

In program verification, one of the fundamental technical open problems is the automatic generation of invariants. Given adequate invariants (in terms of pre/post-conditions, loop invariants, etc.), the rest of the verification process can often be completely automated [18, 25] using logical constraint-solvers [8, 14].

For the purposes of this article, let us consider simple while-programs with a single while-loop. Given a pre-condition, a post-condition, assertions, and contracts for functions called, the problem is to find a loop invariant that will prove the post-condition and assertions (assuming the program is correct).

The natural way to phrase this problem using ALFs is to model the concept space to consist of all subsets of program configurations and to have the hypothesis space to be the set of all logical formulas that capture the class of invariants from which we want to synthesize. For any program, the target specification would be the set $\mathcal{T}_{Inv}$ of all inductive invariants that prove the post-condition and assertions correct. We now describe several frameworks that use such learning. As in most applications, the sample-space and the teacher have to be *co-designed* in order for teachers to exist. Also, the hypothesis space varies from framework to framework.

Invariant synthesis using the ICE learning model Given a program with a single loop whose loop invariant we want to synthesize, there are *many* inductive invariants that prove the assertions in the program correct— these invariants are characterized by the following three properties: (a) that it include the states when the loop is entered the first time, (b) that it exclude the states that immediately exit the loop and reach the end of the program and not satisfy the post-condition, and (c) that it is inductive (i.e., from any state satisfying the invariant, if we execute the loop body once, the resulting state is also in the invariant). The teacher knows these properties, and must reply to conjectured hypotheses of the learner using these properties. Violation of properties (a) and (b) are usually easy to check using a constraint solver, and will result in a *positive* and *negative* concrete configuration as a sample, respectively. However, when inductiveness fails, the obvious counterexample is a *pair* of configurations, (x, y), where x is in the hypothesis but y is not, and where the program state x evolves to the state y across one execution of the loop body.

The work by Garg et. al. [20] hence proposes what they call the *ICE model* (for implication counterexamples), where the learner learns from positive, negative, and implication counterexamples. The author's claim is that without implication counterexamples, the teacher is stuck when presented a hypothesis that satisfies the properties of being an invariant save the inductiveness property.

From the described components we build an ALF $\mathcal{A}_{\text{ICE}} = (\mathcal{C}, \mathcal{H}, \gamma, \mathcal{S}, \kappa)$, 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 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 (i.e., $(P, N, I) \sqsubseteq_{\text{s}} (P', N', I')$ if $P \subseteq P'$, $N \subseteq N'$, and $I \subseteq I'$).
- The join is the component-wise union, and $\bot_{\text{s}} = (\emptyset, \emptyset, \emptyset)$.

Since this sample space contains implications in addition to the standard positive and negative examples, we refer to it as an *ICE sample space*.

We can now show that there is a teacher for these ALF instances, because a teacher can refute any hypothesis made by a learner with a positive, negative, or implication counterexample, depending on which property of invariants is violated.

Proposition 4.1. *There is a teacher for ALF instances of the form* $(\mathcal{A}_{\text{ICE}}, \mathcal{T}_{Inv})$.

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.

Lemma 4.2. *Let* $C = H = 2^D$, $\gamma = id$, $S = \{(P, N) \mid P, N \subseteq D\}$, *and* $\kappa((P, N)) = \{R \subseteq D \mid P \subseteq R \wedge R \cap N = \emptyset\}$.

Let $\mathcal{T} \subseteq C$ *be a target. If there exists a teacher for* $\mathcal{T}$, *then there must exists sets* $B, G \subseteq D$ *such that* $\mathcal{T} = \{R \subseteq D \mid B \cap R = \emptyset$ *and* $G \subseteq R\}$.

Proof. Assume that there exists a teacher for the target set $\mathcal{T}$, and let G and B be the union of all positive examples and the union of all negative examples, respectively, that the teacher returns. Now, we claim that $\mathcal{T} = \{R \subseteq D \mid B \cap R = \emptyset \text{ and } R \subseteq G\}$. Towards a contradiction, assume that this is not the case. Then, there exists an $R \in \mathcal{T}$ such that $B \cap R \neq \emptyset$ or $G \not\subseteq R$. If $B \cap R \neq \emptyset$, then there is some $b \in R$ that was returned as a negative counterexample for some hypothesis. Since $R \in \mathcal{T}$ is not consistent with this negative example b, this contradicts the requirement that the teacher is honest. Similarly, if $G \not\subseteq R$, then there is some $g \in G$ that was returned as a positive counterexample, which contradicts the teacher's honesty. □

The above proves that positive and negative samples are not sufficient for learning invariants, as invariants cannot be defined as all sets that exclude a set of states and include a set of states.

There are several ICE-based invariant synthesis formalisms that we can capture. First, Garg et. al. [20] have considered arithmetic invariants over a set of integer variables $x_1, \ldots, x_\ell$ of the form $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \left(\sum_{k=1}^{\ell} a_k^{i,j} x_k \leq c^{i,j}\right)$, $a_k^{i,j} \in \{-1, 0, 1\}$, where the learner is implemented using a constraint solver that finds smallest invariants that fit the sample. This is accurately modeled in our framework, as in the ICE formulation above, with the hypotheses space being the set of all formulas of this form. The fact that Garg et. al.'s learner produces smallest invariants makes it an Occam learner in the sense of Section 3.2.2 and, hence, it converges in finite time. The approach proposed by Sharma and Aiken [44], C2I, is also an ICE-learner, except that the learner uses *stochastic search* based on a Metropolis Hastings MCMC (Markov chain Monte Carlo) algorithm, which again can be seen as an ALF.

We can also see the work by Garg et. al. [19] on synthesizing quantified invariants for linear data structures such as lists and arrays as ALFs. This framework can infer quantified invariants of the form

$$\forall y_1, y_2 \colon (y_1 \leq y_2 \leq i) \Rightarrow a[y_1] \leq a[y_2].$$

However, Garg et. al. do not represent sets of configurations by means of logical formulas (as shown above) but use an automata-theoretic approach, where a special class of automata, called *quantified data automata (QDAs)*, represent such logical invariants; hence, these QDAs form the hypothesis space in the ALF. The sample space there is also unusual: a sample (modeling a program configuration consisting of arrays or lists) is a *set of valuation words*, where each such word encodes the information about the array (or list) for specially quantified pointer variables pointing into the heap, and where data-formulas state conditions of the keys stored at these locations.

ALFs can also capture the ICE-framework described by Neider [37], where invariants are learned in the context of *regular model-checking* [10]. In regular model checking, program configurations are captured using (finite—but unbounded—or even infinite) words, and sets of configurations are captured using finite automata. Consequently, the hypothesis space is the set of all DFAs (over an a priori chosen, fixed alphabet), and the sample space is an ICE-sample consisting of configurations modeled as words. The learner proposed by Neider constructs consistent DFAs of minimal size and, hence, is an Occam learner that converges in finite time (cf. Section 3.2.2).

We now turn to two other invariant-generation frameworks that skirt the ICE model.

Houdini The Houdini algorithm [17] is a learning algorithm for synthesizing invariants that avoids learning from ICE samples. Given a finite set of *predicates* P, Houdini learns an invariant that is expressible as a conjunction of some subset of predicates (note that the hypothesis space is finite but exponential in the number of predicates). Houdini learns an invariant in time *polynomial* in the number of predicates (and in linear number rounds) and is implemented in the Boogie program verifier [7]. It is widely used (for example, used in verifying device drivers [32, 33] and in race-detection in GPU kernels [9].

The setup here can be modeled as an ALF: we take the concept space $\mathcal{C}$ to be all subsets of program configurations, and the hypothesis space $\mathcal{H}$ to be the set of all conjunctions of subsets of predicates in P, with the map γ mapping each conjunctive formula in $\mathcal{H}$ to the set of all configurations that satisfy the predicates mentioned in the conjunction. We take the sample space to be the ICE sample space, where each sample is a valuation v over P (indicating which predicates are satisfied) and where implication counterexamples are pairs of valuations.

The Houdini learning algorithm itself is the classical conjunctive learning algorithm for positive and negative samples (see [28]), but its mechanics are such that it works for ICE samples as well. More precisely, the Houdini algorithm always creates the *semantically smallest* formula that satisfies the sample (it hence starts with a conjunction of all predicates, and in each round "knocks off" predicates that are violated by positive samples returned). Since Houdini always returns the semantically-smallest conjunction of predicates, it will never receive a negative counterexample (assuming the program is correct and has a conjunctive invariant over P). Furthermore, for an implication counterexample (v, v'), the algorithm knows that since it proposed the semantically smallest conjunction of predicates, v cannot be made negative; hence it treats v' as a positive counterexample. Houdini converges since the hypothesis space is finite (matching the first recipe for convergence we outlined in Section 3.2.1); in

fact, it converges in linear number of rounds since in each round at least one predicate is removed from the hypothesized invariant.

Learning invariants in regular model-checking using witnesses The learning-to-verify project reported in [50–53] leverages machine learning to the verification of infinite state systems that result from processes with FIFO queues, but skirts the ICE model using a different idea. The key idea is to view the the identification of the reachable states of such a system as a machine learning problem instead of computing this set iteratively (which, in general, requires techniques such as acceleration or widening). In particular, we consider the work of Vardhan et. al. [52] and show that this is an instantiation of our abstract learning framework. The key idea in this work is to represent configurations as traces through the system and to add a notion of *witness* to this description, resulting in so-called annotated traces. The teacher, when receiving a set of annotated traces, can actually check whether the configurations are reachable based on the witnesses (a witness can be, say, the length of the execution or the execution itself) and, consequently, the model allows learning from such traces directly. Indeed, this approach can be modeled by an ALF: the concept space consists of subsets of configurations, the target space consists of the set of reachable configurations, the hypothesis space consists of automata over annotated traces, and the sample space consists of positive and negatively labeled annotated traces.

4.1.2 Synthesis of Fixpoints in Abstract Domains

In Section 4.1.1 we have explained how several instances of ICE learners fit into our framework. We now provide a generic technique to model the problem of fixpoint computation in the setting of abstract interpretation using learning.

We assume the setting of abstract interpretation [13] with

- a concrete domain $(\mathcal{D}, \subseteq, \bot, \cup)$ and an abstract domain $(\widehat{\mathcal{D}}, \sqsubseteq, \bot, \sqcup)$, which both have a join lattice structure, and
- a Galois connection between these two, given by two monotone functions $\gamma : \widehat{\mathcal{D}} \to \mathcal{D}$ (the concretization function), and $\alpha : \mathcal{D} \to \widehat{\mathcal{D}}$ (the abstraction function), with $\alpha(X) \sqsubseteq \widehat{X} \Leftrightarrow X \subseteq \gamma(\widehat{X})$ for all $X \in \mathcal{D}$ and $\widehat{X} \in \widehat{\mathcal{D}}$.

The concrete domain usually describes the semantics of a program and comes with an increasing transformer $F : \mathcal{D} \to \mathcal{D}$ that captures the behaviour of the program (increasing means that $X \subseteq X$ for each X). The abstract domain is used to model the aspects of the program that one is interested in.

A specification is given in terms a set $B \subseteq \mathcal{D}$ of bad concrete elements. The goal is to find a fixpoint of F that is not above any bad element, i.e., an element $X \in \mathcal{D}$ with $F(X) = X$, and $Y \not\subseteq X$ for each $Y \in B$. We refer to such fixpoints as *adequate fixpoints* because they show that the program cannot reach a bad state.

Synthesis of Precise Abstract Transformers The general idea of abstract interpretation is to do the fixpoint computation on the abstract domain instead of the concrete domain. This is done using an abstract transformer $\widehat{F} : \widehat{\mathcal{D}} \to \widehat{\mathcal{D}}$ that overapproximates the concrete transformer, in the sense that $\alpha(F(\gamma(\hat{X}))) \sqsubseteq \hat{F}(\hat{X})$ for each $\hat{X} \in \widehat{\mathcal{D}}$. The best abstract transformer is the one where equality holds instead of inclusion.

Since manually designing a good abstract transformer can be difficult, Thakur et al [48] propose an automatic method to synthesize the abstract post of a given abstract element for the concrete domain consisting of sets of program configurations $\mathcal{D} = 2^D$.

We can model the algorithm proposed by Thakur et al [48] as a synthesis using learning in an ALF that uses the concrete domain $\mathcal{D}$ as concept space, the abstract domain $\widehat{\mathcal{D}}$ as hypothesis space together with the existing concretization function γ as the concretization function for the ALF. Since we are interested in elements above $\alpha(F(\gamma(\hat{X})))$, the sample space simply consists of positive examples, that is, a sample is a finite set of elements of D. A hypothesis is consistent with a sample if it contains all the elements from the sample.

For an abstract element $\hat{X}$, the target specification is given by $\mathcal{T} = \{T \subseteq D \mid \alpha(F(\gamma(\hat{X}))) \sqsubseteq T\}$.

If the learner proposes some hypothesis H (which is an abstract element), then the teacher can check whether $\gamma(H)$ is a superset of $F(\gamma(\widehat{X}))$ and if not, return a positive example from $F(\gamma(\widehat{X})) \setminus \gamma(H)$. This is precisely what happens (at an abstract level) in the procedure proposed in [48].

One should note here that in case the abstract domain has a maximal element, a learner could always propose this element because it can be sure that it is a target element if the target is realizable. The idea is to come up with a learner that computes a better solution than just the maximum. In [48] this is done by starting with the empty hypothesis, and then for each positive example S returned by the teacher, applying the abstraction function α on it and taking the join of the resulting abstract element with the current hypothesis. In this way, the learner ensures that the hypothesis is always below the set $\alpha(F(\gamma(\widehat{X})))$. So if the learner converges, then it computes the best abstract transformer on $\widehat{X}$.

Synthesis of Adequate Fixpoints. We now describe a generic way for designing an ALF that models the problem of finding an adequate fixpoint as a synthesis problem that uses learning from samples. This can be seen as a generalization of finding loop invariants in an abstract domain (but without computing the abstract transformers).

The ALF we propose has the concrete domain $\mathcal{D}$ as concept space, the abstract domain $\widehat{\mathcal{D}}$ as hypothesis space together with the existing concretization function γ. The sample space of an ALF is, in general, designed for a specific class of target specifications (in order to guarantee the existence of a teacher). In the following, we describe how to construct such a sample space for adequate fixpoints as targets.

The invariants defined in Section 4.1.1 can be seen as adequate fixpoints for the transformer F and the set B being defined pointwise on single configurations (with concrete domain consisting of sets of program configurations). This pointwise definition is the reason why the sample space can be build up by using single configurations as positive and negative examples, and pairs of configurations as implications.

We replace the pointwise definition by a definition of in terms of a class $\mathcal{R} \subseteq \mathcal{D}$ of representative concrete elements. So the definition below is satisfied by the class of singleton sets in case of the concrete domain consisting of sets of program configurations. It intuitively states that $\mathcal{R}$ is rich enough to prove that a hypothesis is not a fixpoint.

Definition 4.3. *A set $\mathcal{R} \subseteq \mathcal{D}$ is a* representative set *(for the transformer F) if for each $\hat{X} \in \widehat{\mathcal{D}}$ and $X := \gamma(\hat{X})$ such that $F(X) \neq X$, there are $Y, Y' \in \mathcal{R}$ with $Y \subseteq X$, $Y' \subseteq F(Y)$, and $Y' \not\subseteq X$.* □

It is not hard to see that $\mathcal{D}$ is always representative set. And in fact that the set of all elements of the form $\gamma(H)$ and $F(\gamma(H))$, where $H \in \mathcal{H}$ also forms a representative set.

We consider target specifications of adequate fixpoints that can be expressed in terms of $\mathcal{R}$. We say that $\mathcal{T} \subseteq \mathcal{D}$ is an $\mathcal{R}$-specification of adequate fixpoints if there is a set $B \subseteq \mathcal{R}$ such that $\mathcal{T} = \{X \in \mathcal{D} \mid F(X) = X \text{ and } Y \not\subseteq X \text{ for all } Y \in B\}$.

For this class of target specifications, we let the sample space $\mathcal{S}_\mathcal{R}$ consist of ICE samples over $\mathcal{R}$, that is, of triples (P, N, I) of finite set $P, N \subseteq \mathcal{R}$, and a finite set $I \subseteq \mathcal{R} \times \mathcal{R}$. A concept $X \in D$ is consistent with a sample (P, N, I) if $Y \subseteq X$ for all $Y \in P$, $Y \not\subseteq X$ for all $Y \in N$, and if $Y \subseteq X$, then $Y' \subseteq X$ for all $(Y, Y') \in I$. This consistency relation is denoted by $\kappa_\mathcal{R}$. In analogy to Proposition 4.1 we can show that this sample space is expressive enough to guarantee the existence of a teacher.

Proposition 4.4. *There is a teacher for ALF instances of the form $(\mathcal{A}, \mathcal{T})$ with $\mathcal{A} = (\mathcal{D}, \widehat{\mathcal{D}}, \gamma, \mathcal{S}_\mathcal{R}, \kappa_\mathcal{R})$ and an $\mathcal{R}$-specification of adequate fixpoints $\mathcal{T}$.*

This provides a generic way of setting up a learning scenario for abstract interpretation, and thus provides a powerful tool for understanding the requirements for the application and development of machine learning algorithms for the synthesis of adequate fixpoints.

4.2 Program Synthesis

In this section, we study several examples of learning-based program synthesis, which include synthesizing program expressions, expressions to be plugged in program sketches, snippets of programs, etc., and show how they can be modeled as ALFs.

End-user synthesis from examples: Flashfill One application of synthesis is to use it to help end-users to program using examples. A prime example of this is FLASHFILL by Gulwani et al [22], where the authors show how string manipulation macros from user-given input-output examples can be synthesized in the context of Microsoft Excel spreadsheets. Flashfill can be seen as an ALF: the concept space consists of all functions from strings to strings, the hypothesis space consists of all string manipulation macros, and the sample space consists of a sets of input-output examples for such functions. The consistency relation κ maps each sample to all functions that agree with the sample. The role of the teacher is played by the *user*: the user has some function in mind and gives new input-output examples whenever the learner returns a hypothesis that she is not satisfied with. The learning algorithm here is based on version-space algebras (which, intuitively, compactly represents *all* possible macros with limited size that are consistent with the sample) and in each round proposes a simple macro from this collection.

Completing sketches and the SyGuS solvers The sketch-based synthesis approach [45] is another prominent synthesis application, 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 here 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 $\exists \vec{e}. \forall \vec{x} \psi(\vec{e}, \vec{x})$, where $\vec{e}$ are the expressions to synthesize and $\vec{x}$ are the inputs to the program.

The Sketch system works by (a) unfolding loops a finite number of times, hence, bounding the length of executions, and (b) encoding the choice of expressions $\vec{e}$ to be synthesized using bits (typically using templates and representing integers by a small number of bits). For the synthesis step, the Sketch system implements a CEGIS (counterexample guided synthesis) technique using SAT solving, whose underlying idea is to learn the expressions from examples using only a SAT solver. The CEGIS technique works in rounds: the learner proposes hypothesis expressions and the teacher checks whether $\forall \vec{x} \psi(\vec{e}, \vec{x})$ holds (using SAT queries) and if not, returns a valuation for $\vec{x}$ 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 $\psi(\vec{e}, \vec{x})$ holds for every valuation of $\vec{x}$ returned by the teacher thus far; the resulting expressions are the hypotheses for the next round. Note that the use of samples avoids quantifier alternation both in the teacher and the learner.

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 $\psi(\vec{e}, \vec{v})$ where the variables $\vec{x}$ have been substituted with a concrete valuation. The relation κ maps such a sample to the set of all expressions $\vec{f}$ such that the formulas in the sample all evaluate to true if $\vec{f}$ is substituted for $\vec{e}$. 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. Since expressions are encoded by a finite number of bits, the hypothesis space is finite, and the Sketch learner converges in finite time (cf. Section 3.2.1).

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 $(G, \psi(\vec{f}))$ where G is a grammar that imposes syntactic restrictions for functions (or expressions) written using symbols of the background theory, and ψ is a formula, again in the theory T, including function symbols $\vec{f}$; the functions $\vec{f}$ are typed according to domains of T. The goal is to find functions $\vec{g}$ for the symbols $\vec{f}$ in the syntax G such that ψ holds. The competition version further restricts ψ to be of the form $\forall \vec{x} \psi'(\vec{f}, \vec{x})$ where ψ' is a quantifier-free formula in a decidable SMT theory—this way, given a hypothesis for the functions $\vec{f}$, the problem of checking whether the functions meet the specification is decidable.

There have been several solvers developed for SyGuS (cf. the first SyGuS competition [2, 3]), and all of them are in fact learning-based (i.e., CEGIS) techniques. 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 possible 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). All three solvers work by generating a tuple of functions such that $\forall \vec{x} \psi'(\vec{f}, \vec{x})$ holds for all valuations of $\vec{x}$ given by the teacher thus far. The enumerative solver enumerates functions until it reaches such a function, the stochastic solver searches the space of functions randomly using a measure that depends on how many samples are satisfied till it finds one that satisfies the samples, and the constraint-based solver queries a constraint-

solver for instantiations of template functions so that the specification is satisfied on the sample valuations. Both the enumerative and the constraint-solver are Occam learners and, hence, converge in finite time.

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 $\psi'(\vec{f}, \vec{v})$ consisting of the specification with particular values $\vec{v}$ substituted for $\vec{x}$. The learners can now be seen as learning from these samples, without knowledge of ψ (similar to the modeling of 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 (see Section 4.1.1) 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 intermediate states in the execution of the loop.

Also, recent work [42] explores the synthesis of *guarded affine functions* from a sample space that consists of information of the form $f(\vec{s}) = t$, where $\vec{s}$ and $\vec{t}$ are integers. The learner here uses a combination of computational geometry techniques and decision tree learning, and can also be modeled as an ALF. Notice that this sample space precisely matches the sample space for deobfuscation problems (where the teacher can return counterexamples of this form – see Example at the end of Section 2 on page 5 – using the program being deobfuscated). Consequently, the learner in Alchemist [42] can be used for deobfuscating programs that compute guarded affine functions from tuples of integer inputs to integers (like the "multiply by 45" example in [26]).

Other synthesis engines There are several algorithms that are self-described as CEGIS frameworks, and, hence, can be modeled using ALFs. For example, synthesizing loop-free programs [23], synthesizing synchronizing code for concurrent programs [11] (in this work, the sample space consists of abstract concurrent partially-ordered traces), work on using synthesis to *mine specifications* [27], synthesizing bit-manipulating programs and deobfuscating programs [26] (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 [43], deductive program repair [30], synthesis of recursive functional programs over unbounded domains [29], as well as synthesis of protocols using enumerative CEGIS techniques [49].

5. Variations and Limitations of the Framework

In this section we discuss some variations and limitations of our framework. We start by briefly discussing a variation of our framework that omits the concept space.

5.1 Omitting the Concept Space

We believe that, for a clean modeling of a synthesis problem, one should specify the concept space $\mathcal{C}$. This makes it possible to compare different synthesis approaches that work with different representations of hypotheses and maybe different types of samples over the same underlying concept space.

However, for the actual learning process, the concept space itself is not of great importance because the learner proposes elements from the hypothesis space, and the teacher returns an element from the sample space. The concept space only serves as a semantic space that gives meaning to hypotheses (via the concretization function γ), and to the samples (via the consistency relation κ).

Therefore, it is possible to omit the concept space from an ALF, and to directly specify the consistency of samples with hypotheses. Such a reduced ALF would then be of the form $\mathcal{A} = (\mathcal{H}, \mathcal{S}, \kappa)$ with a function $\kappa : \mathcal{S} \to 2^{\mathcal{H}}$. In the original framework, this corresponds to the function $\kappa_{\mathcal{H}}$ defined by $\kappa_{\mathcal{H}}(S) = \gamma^{-1}\kappa(S)$.

To create ALF instances, the target specification is also directly given as a subset of the hypothesis space $\mathcal{T} \subseteq \mathcal{H}$. All the other definitions can be adapted directly to this framework.

5.2 Limitations

The ALF framework we develop in this paper is not meant to capture every existing method that uses learning from samples. There are several synthesis techniques that use grey-box techniques (a combination of black-box learning from samples and by utilizing the specification of the target directly in some way) or use query models (where they query the teacher for various aspects of the target set).

For instance, there are active iterative learning scenarios in which the learner can ask other types of questions to the teacher than just proposing hypotheses that are then accepted or refuted by the teacher. One prominent scenario of this kind is Angluin's active learning of DFAs [4], where the learner can ask *membership queries* and *equivalence queries*. (The equivalence queries correspond to proposing a hypothesis, as in our framework, which is then refuted with a counterexample if it is not correct.) Such learning scenarios for synthesis are used, for example, in [1] for the synthesis of interface specifications for Java classes, and in [39] for automatically synthesizing assumptions for assume-guarantee reasoning. Our framework does not have a mechanism for directly modeling such queries. The ALF framework that we have presented is intentionally a simpler framework by design that captures and cleanly models emerging synthesis procedures in the literature where the learner only proposes hypotheses and learns from samples the teacher provides in terms of samples to show that the hypothesis is wrong. The learner in our framework, being a completely passive learner (as opposed to an active learner), can also be implemented by the variety of scalable passive machine-learning algorithms in vogue [36]. Clean extension of ALFs to query settings and grey-box settings would be an interesting future direction to pursue.

6. 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 were motivated by abstract interpretation [13] 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 synthe-

sis that generalizes and extends existing techniques (again, in a similar vein as to how widening and narrowing in abstract interpretation gives 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.

References

[1] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109. ACM, 2005. . URL `http://doi.acm.org/10.1145/1040305.1040314`.

[2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FM-CAD 2013*, pages 1–8. IEEE, 2013.

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

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] D. Angluin. Computational learning theory: Survey and selected bibliography. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 351–369. ACM, 1992. ISBN 0-89791-511-9.

[6] A. Baker. Simplicity. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2013 edition, 2013. `http://plato.stanford.edu/archives/fall2013/entries/simplicity/`.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[9] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 113–132. ACM, 2012.

[10] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV 2000*, volume 1855, pages 403–418. Springer, 2000.

[11] P. Cerny, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis, 2015. CAV 2015, to appear.

[12] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. . URL `http://dx.doi.org/10.1007/BF00994018`.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[14] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[15] P. M. Domingos. The role of occam's razor in knowledge discovery. *Data Min. Knowl. Discov.*, 3(4):409–425, 1999. . URL `http://dx.doi.org/10.1023/A:1009868929893`.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 449–458. ACM, 2000.

[17] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[18] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. AMS, 1967.

[19] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV 2013*, volume 8559, pages 813–829. Springer, 2013.

[20] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV 2014*, volume 8559, pages 69–87. Springer, 2014.

[21] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. Technical report, University of Illinois at Urbana-Champaign, 2015. http://hdl.handle.net/2142/77025.

[22] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL 2011*, pages 317–330. ACM, 2011.

[23] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011.

[24] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proc. London Math. Soc.*, s3-2(1):326–336, 1952. . URL `http://dx.doi.org/10.1112/plms/s3-2.1.326`.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010.

[27] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*, pages 43–52. ACM, 2013.

[28] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

[29] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 407–426. ACM, 2013.

[30] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV), to appear*, 2015.

[31] V. Kuncak. Verifying and synthesizing software with recursive functions - (invited contribution). In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2014.

[32] A. Lal and S. Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 202–212. ACM, 2014.

[33] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 427–443. Springer, 2012.

[34] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987. . URL `http://dx.doi.org/10.1007/BF00116827`.

[35] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2 (1):90–121, Jan. 1980. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/357084.357090`.

[36] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

[37] D. Neider. *Applications of Automata Learning in Verification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

[38] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015.

[39] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008. . URL `http://dx.doi.org/10.1007/s10703-008-0049-6`.

[40] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Symposium on Principles of Programming Languages, POPL'89*, pages 179–190, 1989.

[41] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN 1-55860-238-0.

[42] S. Saha, P. Garg, and P. Madhusudan. Alchemist: Learning guarded affine functions, 2015. CAV 2015, to appear.

[43] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316. ACM, 2013.

[44] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2014.

[45] A. Solar-Lezama. *Program Synthesis by Sketching″*. PhD thesis, University of California at Berkeley, 2008.

[46] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[47] A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, University of Wisconsin, Madison, WI, Apr 2013.

[48] A. V. Thakur, A. Lal, J. Lim, and T. W. Reps. Posthat and all that: Automating abstract interpretation. *Electr. Notes Theor. Comput. Sci.*, 311:15–32, 2015. . URL `http://dx.doi.org/10.1016/j.entcs.2015.02.003`.

[49] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296. ACM, 2013.

[50] A. Vardhan and M. Viswanathan. Learning to verify branching time properties. In *ASE 2005*, pages 325–328. ACM, 2005.

[51] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In *FSTTCS 2004*, volume 3328, pages 494–505. Springer, 2004.

[52] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *ICFEM 2004*, volume 3308, pages 274–289. Springer, 2004.

[53] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *TACAS 2005*, volume 3440, pages 45–60. Springer, 2005.