Synthesizing Piece-wise Functions by Learning Classifiers

Daniel Neider[1,2], Shambwaditya Saha[1], and P. Madhusudan[1]

[1] University of Illinois at Urbana-Champaign
[2] University of California, Los Angeles

Abstract. We present a novel general technique that classifier learning to synthesize piece-wise functions (functions that split the domain into *regions*, applying simpler functions to each region), working in combination with a synthesizer of the simpler functions for concrete inputs and a synthesizer of predicates that can be used to define regions. We develop a theory of single-point refutable specifications that facilitate generating concrete counterexamples using constraint solvers. We implement the framework for synthesizing piece-wise functions in linear arithmetic, combining leaf expression synthesis using constraint-solving and predicate synthesis using enumeration, and tie them together using a decision tree classifier. We demonstrate that this approach is competitive compared to existing synthesis engines on a set of specifications.

1 Introduction

The field of synthesis is an evolving discipline in formal methods that is seeing a renaissance, mainly due to a variety of new techniques [1] to automatically synthesize small expressions or programs that are useful in niche application domains, including end-user programming [12], filling holes in program sketches [26], program transformations [15, 6], automatic grading of assignments [2, 24], synthesizing network configurations and migrations [23, 17], synthesizing annotations such as invariants or pre/post conditions for programs [10, 11], etc.

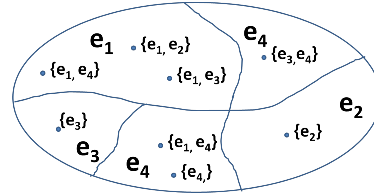
The field of machine learning [18] is close to program synthesis, especially when the specification is a set of input-output examples. The subfield of inductive programming has a long tradition in solving this problem using inductive methods that generalize from the sample to obtain programs [16]. Machine learning, which is the field of learning algorithms that can predict data from training data, is a rich field that encompasses algorithms for several problems, including classification, regression, clustering, etc.

The idea of using inductive synthesis for more general specifications than input-output examples has been explored extensively in program synthesis research. The counterexample guided inductive synthesis (CEGIS) approach to program synthesis advocates pairing inductive learning algorithms with a verification oracle: in each round, the learner learns inductively from a set of (counter-)examples and proposes an expression which the verification oracle checks against the specification, and augments the set of samples with a new counterexample [26].

A majority of the current synthesis approaches rely on counter-example guided inductive synthesis [26, 10, 11, 13].

In this paper, we consider logical specifications for synthesis, where the goal of synthesis is to find some expression e for a function f, in a particular syntax, that satisfies a specification $\forall \vec{x}.\psi(\vec{x})$.[3] We will assume that ψ is quantifier-free, that the satisfiability of the quantifier-free theory of the underlying logic is decidable, and that there is an effective algorithm that can also produce models. The goal of this paper is to develop a new framework for expression synthesis that can learn *piece-wise functions* using a *learning algorithm for classifiers* with the help of two other synthesis engines, one for synthesizing expressions for *single* inputs and another for synthesizing predicates that separate concrete inputs from each other. The framework is general in the sense that it is independent of the logic used to write specifications and the logic used to express the synthesized expressions.

A piece-wise function is a function that partitions the input domain into a finite set of regions, and then maps each region using a simpler class of functions. The framework that we build for expression synthesis is also counterexample guided, and proceeds in the following fashion (see Figure 1 on Page 9 and see figure on right):



- In every round, the learner proposes a piece-wise function H for f, and the verification oracle checks whether it satisfies the specification. If not, it returns one input $\vec{p}$ on which H is incorrect. (Returning such a counterexample is nontrivial; we will discuss this issue below.)
- We show that we can now use an *expression synthesizer* for the single input $\vec{p}$ which synthesizes an expression that maps $\vec{p}$ to a correct value. This expression synthesizer will depend on the underlying theory of basic expressions, and we can use any synthesis algorithm that performs this task.
- Once we have the new expression, we compute for every counterexample input obtained thus far the set of basic expressions synthesized so far that work correctly for these inputs. This results in a set of *samples*, where each sample is of the form $(\vec{p}, Z)$, where $\vec{p}$ is a concrete input and Z is the set of basic expressions that are correct for $\vec{p}$ (see points with labels in figure above). The problem we need to solve now can be seen as a multi-label classification problem— that of finding a mapping from *every* input to an expression that is consistent with the set of samples.
- Since we want a classification that is a piece-wise function that divides the input domains into regions, and since the predicates needed to define regions can be arbitrarily complex and depend on the semantics of the underlying logical theory, we require a *predicate synthesizer* that synthesizes predicates that can separate concrete inputs with disjoint sets of labels. Once we have such a set of predicates, we are equipped with an adequate number of regions to find a piece-wise function.

[3] Note that this can, of course, describe input-output examples as well.

- The final phase uses *classification learning*, to generalize the sample to a function from all inputs to basic expressions (see figure above). The learning should be biased towards finding *simpler* functions, finding fewer regions or minimizing the Boolean expression that describes the piece-wise function.

The framework above requires many components, in addition to the expression synthesizer and predicate synthesizer. For example, given a hypothesis function H and a specification $\forall \vec{x}. \psi(f, \vec{x})$, we need to find a concrete counterexample input on which H is wrong. It turns out that there may be *no* such point in some specifications and even if there was, synthesizing one may be hard. We develop a theory of *single-point definable specifications* that ensure such counterexample inputs exist, and a subclass of *single-point refutable specifications* that reduce finding such counterexamples to satisfiability problems of the underlying logical domain (which is decidable). Our framework works robustly for the class of singe-point refutable specifications, and we show how to extract concrete counterexamples, how to automatically synthesize a new specification tailored for any input $\vec{p}$ to be given to the expression synthesizer, and how to evaluate whether particular expressions work for particular inputs.

In current standard CEGIS approaches [26, 1], when H and $\forall \vec{x}. \psi(f, \vec{x})$ are presented, the teacher simply returns a concrete value of $\vec{x}$ for which $\neg\psi(H/f, \vec{x})$ is satisfied. We emphasize that such valuations for the universally quantified variables cannot be interpreted as inputs on which H is incorrect, and hence cannot be used in our framework. The framework of single-point refutable specifications is crucial in using classifiers to synthesize expressions.

The classifier learning algorithm can be any learning algorithm for multi-label classification, with the learning bias as described above, but must ensure that the learned classifier is *consistent* with the given samples. Machine-learning algorithms more often than not make mistakes and are not consistent with the sample, often because they want to generalize assuming that the sample could be noisy. In Section 5, we describe the second contribution of this paper—an adaptation of decision-tree learning to multi-label learning that produces classifications consistent with the sample, and explore a variety of statistical measures used within the decision-tree learning to bias the learning towards smaller trees in the presence of multi-labeled samples. The resulting decision-tree learning algorithms form one class of classifiers that can be used to synthesize piece-wise functions over any theory that works using our framework.

The third contribution of the paper is an instantiation of our framework to build a state-of-the-art synthesizer of piece-wise linear arithmetic functions for specifications given in the theory of linear arithmetic. We implement the components of the framework for single-point refutable functions: to synthesize input counterexamples, to reformulate the synthesis problem for a single input, and to evaluate whether an expression works correctly for any single input. These problems are reduced to the satisfiability of the underlying quantifier-free theory of linear integer arithmetic, which is decidable using SMT solvers. The expression-synthesizer for single inputs is performed using an inner CEGIS-based engine based on constraint-solvers.The predicate synthesizer is instantiated using an enumerative synthesis algorithm. The resulting solver works extremely well

on a large class of benchmarks drawn from a recent SyGuS synthesis competition (linear-arithemetic track) where a version of our solver fared significantly better than all the traditional SyGuS solvers (enumerative, stochastic, and symbolic constraint-based solvers). In our experience, finding an expression that satisfies a single input is a much easier problem for current synthesis engines (where constraint solvers that compute the coefficients defining such an expression are very effective) than finding one that satisfies all inputs. The decision-tree based classification on the other hand solves the problem of generalizing this labeling to the entire input domain effectively.

Related Work Our learning task is closely related to the syntax-guided synthesis framework (SyGuS) [1], which provides a language, similar to SMTLib [4], to describe synthesis problems. SyGuS is more general than our setting: the synthesis tasks can be parameterized by a background theory, it can involve more than one function, and syntactic restriction can be imposed on the functions to synthesize. Several solvers for SyGuS have been developed [1], including an enumerative solver, a solver based on constraint solving, one based on stochastic search, and one based on the program synthesizer Sketch [25]; it is worth noting that all solvers follow the counterexample-guided inductive synthesis approach (CEGIS) [26]. Recently, a solver based on CVC4 [21] has also been presented.

There has been several work on synthesizing piece-wise affine models of hybrid dynamical systems from input-output examples [3, 5, 9, 28] (we refer the reader to [19] for a comprehensive survey). The setting there is to learn an affine model passively (i.e., without feedback whether the synthesized model satisfies some specification) and, consequently, only approximates the actual system. A tool for learning guarded affine functions, which uses a CEGIS approach, is Alchemist [22]. In contrast to the setting presented in this work, however, Alchemist requires that the function to synthesize is unique.

2 The Synthesis Problem and Single-point Refutable Specifications

The synthesis problem we tackle in this paper is that of finding a function f that satisfies a logical specification of the form $\forall \vec{x}.\ \psi(f, \vec{x})$, where ψ is a *quantifier-free* first-order formula over a logic with fixed interpretations of constants, functions, and relations. Further, we will assume that the quantifier-free fragment of this logic admits a *decidable* satisfiability problem and furthermore, effective procedures for producing a model that maps the variables to the domain of the logic are available. These effective procedures are required in order to build tractable counter-example generation while performing synthesis.

More formally, let f be function symbol representing the target function that needs to be synthesized and let us assume its arity is n, for the rest of the paper. The specification logic is a formula in first-order logic, over an arbitrary set of function symbols (including f), constants, and relations, all of them with fixed interpretations, except for f. We will assume that the logic is interpreted over a

countable universe D and, further, and that there is a constant symbol for every element in D. For technical reasons, we will assume that negation is pushed all the way in to atomic predicates.

The specification for synthesis is a formula of the form $\forall \vec{x}.\ \psi(f, \vec{x})$ where ψ is a formula expressed in the following grammar:

$$\begin{aligned} \textit{Term} \quad t &::- x \mid c \mid f(t_1, \ldots, t_n) \mid g(\vec{t}\,) \\ \textit{Formula} \quad \varphi &::- P(\vec{t}\,) \mid \neg P(\vec{t}\,) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \end{aligned}$$

We will assume that equality is a relation in the logic, with the standard model-theoretic interpretation.

The synthesis problem is to find a definition for the function f in a particular syntax that satisfies the specification. More formally, we would like to find an *expression* for f such that it satisfies the specification. We will assume that the syntax for expressions is a term with free variables $y_1, \ldots, y_n$ of the form:

$$\textit{Term} \quad t ::- c \mid y_i \mid \widehat{g}(\vec{t}\,) \mid \textit{ite}(\widehat{P}(\vec{t}\,), t_1, t_2)$$

where $\widehat{g}$ and $\widehat{P}$ are (interpreted) functions in the underlying logic, but which can be restricted to a subset of those present in the logic.

Given a specification $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$, the synthesis problem is to find an expression e such that $\forall \vec{x}.\ \psi(e/f,\ \vec{x}\,)$ is valid.

Single-point definable specifications In order to be able to define a general counter-example guided inductive synthesis (CEGIS) algorithm for synthesizing expressions for f based on learning classifiers, as described in Section 1, we need to be able to refute any hypothesis H that doesn't satisfy the specification with a concrete input where H is wrong. We will now define sufficient conditions that guarantee this property. The first is a semantic property, called *single-point definable specifications* that guarantees the existence of such concrete input counterexamples and the second is a syntactic fragment of the former, called *single-point refutable specifications* that allow such concrete counterexamples to be found effectively using a constraint solver.

A single-point definable specification is, intuitively, a specification that restricts how each input is mapped to the output, *independent* of how other inputs are mapped to outputs. More precisely, a single-point definable specification restricts each input $\vec{p} \in D^n$ to a *set of outputs* $X_{\vec{p}} \subseteq D$, and allows any function that respects this restriction for each point. It cannot, however, restrict the output on $\vec{p}$ based on how the function behaves on other inputs. Many synthesis problems fall into this category (see Section 7 for several examples taken from a synthesis competition).

Formally, we define this concept as follows. Let $I = D^n$ be the set of inputs and $O = D$ be the set of outputs of the function being synthesized.

Definition 1. *A specification α is said to be* single-point definable *if the following holds. Let $\mathcal{F}$ be the class of all functions h such that h satisfies the specification α. Let $g : I \to O$ be a function such that for every $\vec{p} \in I$, there exists some $h \in \mathcal{F}$ such that $g(\vec{p}) = h(\vec{p})$. Then $g \in \mathcal{F}$ (i.e., g satisfies the specification α).*

Intuitively, a specification is single-point definable if whenever we construct a function that maps each input according to *some* function that satisfies the specification, the resulting function will itself satisfy the specification. For each input $\vec{p}$, if $X_{\vec{p}}$ is the set of all outputs that meet the specification map $\vec{p}$ to, then any function g that maps every input $\vec{p}$ to some element in $X_{\vec{p}}$ will also satisfy the specification. This captures the requirement, semantically, that the specification constrains the outputs for each input, independent of each other.

For example, the following specifications are all single-point definable specifications over the first-order theory of arithmetic:

- $f(15, 23) = 19 \wedge f(90, 20) = 91 \wedge \ldots \wedge f(28, 24) = 35$.
  More generally, any set of input-output samples can be written as a conjunction of constraints that forms a single-point specification.
- Any specification that is not realizable (has no function that satisfies it).
- $\forall x.(f(0) = 0 \wedge f(x + 1) = f(x) + 1$.
  The identity function is the only function that satisfies this specification. Any specification that has a unique solution is clearly single-point definable.

Note that for any SPD specification $\forall \vec{x}\ \psi(f,\ \vec{x}\ )$, if H is some expression conjectured for f that does not satisfy the specification, there will always be *one* input $\vec{p} \in D^n$ on which H is *definitely wrong* in that no correct solution agrees with H on $\vec{p}$. More precisely, we obtain the following straightforward result.

Lemma 1. *Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a single-point definable specification and let $h : D^n \rightarrow D$ be an interpretation for f such that $\forall \vec{x}.\psi(h/f, \vec{x})$ does not hold. Then there is an input $\vec{p}$ such that for* every *function $h' : D^n \rightarrow D$ that satisfies the specification, $h(\vec{p}) \neq h'(\vec{p})$.*

Single-point refutable specifications We now define single-point refutable specifications, which we will show is a subclass of single-point definable specifications, and for which we can reduce the problem of finding counterexample inputs to a conjectured function to logical satisfiability of the underlying quantifier-free logic.

Intuitively, a specification $\forall \vec{x}\ .\ \psi(f,\ \vec{x}\ )$ is single-point refutable if for any given hypothetical interpretation H to the function f that does not satisfy the specification, we can find a particular input $\vec{p} \in D^n$ such that the formula $\exists \vec{x}.\ \neg\psi(f,\ \vec{x}\ )$ evaluates to false, and where the falsehood is caused *solely* by the interpretation the output H maps $\vec{p}$ to. The definition of single-point refutable specifications is involved as we have to define what it means for H on $\vec{p}$ to solely contribute to falsifying the specification, and involves evaluating the formula $\neg\psi(f,\ \vec{x}\ )$ in a way that ignores the interpretation H on all values except on $\vec{p}$, and checking if it still evaluates to false.

We first define an alternate semantics for a formula $\psi(f,\ \vec{x}\ )$ that is parameterized by a set of n variables $\vec{u}$ denoting an input, and a variable v denoting an output, and a Boolean variable b. The idea is that this alternate semantics evaluates the function by interpreting f on u to be v, but "ignores" the interpretation of f on all other inputs, and reports whether the formula would evaluate to b. We do this by expanding the domain to $D \cup \{\perp\}$, where $\perp$ is a new element,

and have f map all inputs other than $\vec{u}$ to $\bot$. Furthermore, when evaluating formulas, we let them evaluate to b only when we are sure that the evaluation of the formula to b depended only on the definition of f on u. We now define this alternate semantics by *transforming* a formula $\psi(f,\ \vec{x}\,)$ to a formula with the usual semantics, but over the domain $D \cup \{\bot\}$. In this transformation, we will use if-then-else (*ite*) terms for simplicity.

Definition 2 (The Isolate transformer). *Let $\vec{u}$ be a vector of n first-order variables (where n is the arity of the function to be synthesized) and let v another first-order variable, and let $b \in \{T, F\}$.*

Let $D^+ = D \cup \{\bot\}$, where $\bot \notin D$, be the extended domain, and let the functions and predicates be extended to this domain (the precise extension does not matter).

For a formula $\psi(f,\ \vec{x}\,)$, we define the formula $Isolate_{\vec{u},v,b}(\psi(f,\ \vec{x}\,))$ over the extended domain by

$$Isolate_{\vec{u},v,b}(\psi(f,\ \vec{x}\,)) \coloneqq ite\left(\bigvee_{x_i} x_i = \bot, \neg b, Isol_{\vec{u},v,b}(\psi(f,\ \vec{x}\,))\right),$$

where $Isol_{\vec{u},v,b}$ is defined recursively as follows:

- $Isol_{\vec{u},v,b}(x) = x$
- $Isol_{\vec{u},v,b}(c) = c$
- $Isol_{\vec{u},v,b}(g(t_1, \ldots, t_k)) = ite(\ \bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot,\quad \bot,\quad g(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)))$
- $Isol_{\vec{u},v,b}(f(t_1, \ldots, t_n)) = ite(\ \bigwedge_{i=1}^{n} Isol_{\vec{u},v,b}(t_i) = u[i], v, \bot)$
- $Isol_{\vec{u},v,b}(P(t_1, \ldots, t_k)) = ite(\ \bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot,\quad \neg b,\quad P(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)))$
- $Isol_{\vec{u},v,b}(\neg P(t_1, \ldots, t_k)) = ite(\ \bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot,\quad \neg b,\quad \neg P(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)))$
- $Isol_{\vec{u},v,b}(\varphi_1 \vee \varphi_2) = Isol_{\vec{u},v,b}(\varphi_1) \vee Isol_{\vec{u},v,b}(\varphi_2)$
- $Isol_{\vec{u},v,b}(\varphi_1 \wedge \varphi_2) = Isol_{\vec{u},v,b}(\varphi_1) \wedge Isol_{\vec{u},v,b}(\varphi_2)$ □

Intuitively, the function $Isolate_{\vec{u},v,b}(\psi)$ captures the property as to whether ψ will evaluate to b if f maps $\vec{u}$ to v and independent of how f is interpreted on other inputs. A function of the form $f(t_1, \ldots t_n)$ is interpreted to be v if the input matches $\vec{u}$ and otherwise evaluated to $\bot$. This ensures that the uninterpreted function is evaluated only on the single input $\vec{u}$ and all other inputs are sent to $\bot$. Functions on terms that involve $\bot$ are sent to $\bot$ as well. Predicates are evaluated to b only if the predicate is evaluated on terms none of which is $\bot$—otherwise, they get mapped to $\neg b$, to reflect that it will not help to make the final formula ψ evaluate to b. Note that when $Isolate_{\vec{u},v,b}(\psi)$ evaluates to $\neg b$, there is no property of ψ that we claim. Also, note that $Isolate_{\vec{u},v,b}(\psi(f, \vec{x}))$ has no occurrence of f in it, but has free variables $\vec{x}$, $\vec{u}$ and v.

We can show (using a induction over the structure of the specification) that the isolation of a specification to a particular input with $b = F$, when instantiated according to a function that satisfies a specification, cannot evaluate to false (see Appendix A for a proof).

Lemma 2. *Let $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ be a specification and $h \colon D^n \to D$ a function satisfying the specification. Then, there is no interpretation of $\vec{u}$ and $\vec{x}$ such that if v is interpreted as $h(\vec{u})$, the formula $Isolate_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ evaluates to false.*

We can also show (again using a structural induction) that when the isolation of the specification with respect to $b = F$ evaluates to false, then v is definitely not a correct output on $\vec{u}$ (see Appendix B for a proof).

Lemma 3. *Let $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ be a specification, $\vec{p}$ an interpretation for $\vec{u}$, and q an interpretation for v such that the formula $\mathit{Isolate}_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ evaluates to false. Then, there is no function h satisfying the specification that maps $\vec{p}$ to q.*

We can now define single-point refutable specifications.

Definition 3 (Single-point refutable specifications (SPR)). *A specification $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ is said to be* single-point refutable *if the following holds. Let $H : D^n \rightarrow D$ be any interpretation for the function f that does not satisfy the specification (i.e., the specification does not hold under this interpretation for f). Then there exists some input $\vec{p}$ that is an interpretation for $\vec{u}$ and an interpretation for $\vec{x}$ such that when v is interpreted to be $H(\vec{u})$, the isolated formula $\mathit{Isolate}_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ evaluates to false.*

Intuitively, the above says that a specification is single-point refutable if whenever a hypothesis function H does not find a specification, there is a single input $\vec{p}$ such that the specification evaluates to false independent of how the function maps inputs other than $\vec{p}$. More precisely, ψ evaluates to *false* for some interpretation of $\vec{x}$ only assuming that $f(\vec{p}) = H(\vec{p})$.

We can show that single-point refutable specifications are single-point definable, which we formalize below (a proof can be found in Appendix C).

Lemma 4. *If a specification $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ is single-point refutable then it is single-point definable.*

Here are some examples and non-examples of single-point refutable specifications in the first-order theory of arithmetic:

- $f(15, 23) = 19 \wedge f(90, 20) = 91 \wedge \ldots \wedge f(28, 24) = 35$.
  More generally, any set of input-output samples can be written as a conjunction of constraints that forms a single-point refutable specification.
- $\forall x.(f(0) = 0 \wedge f(x+1) = f(x)+1$ is not a single-point refutable specification thought it is single-point definable. Given a hypothesis function (e.g., $H(i) = 0$, for all i), the formula $f(x+1) = f(x)$ evaluates to false, but this involves the definition of f on *two* inputs, and hence we cannot isolate a single input on which the function H is incorrect. (In evaluating the isolated transformation of the specification parameterized with $b = F$, at least one of $f(x+1)$ and $f(x)$ will evaluate to $\perp$ and hence the whole formula never evaluate to *false*.)

When a specification $\forall\ \vec{x}\ .\psi(f,\ \vec{x}\,)$ is single-point refutable, given an *expression* H for f, we can check whether H satisfies the specification by substituting H for f in the specification. Checking validity of this formula (by negating it and checking for satisfiability of the resulting quantifier-free formula) would tell us whether H satisfies the specification. Moreover, if H does not satisfy the specification (which is single-point refutable), it follows that the formula $\exists u.\exists \vec{x}.\mathit{Isolate}_{\vec{u},H(u),F}(\psi(H/f, \vec{x})$ is satisfiable. Assuming the underlying quantifier-free theory has a decidable satisfiability problem, we can find a *concrete* input $\vec{p}$

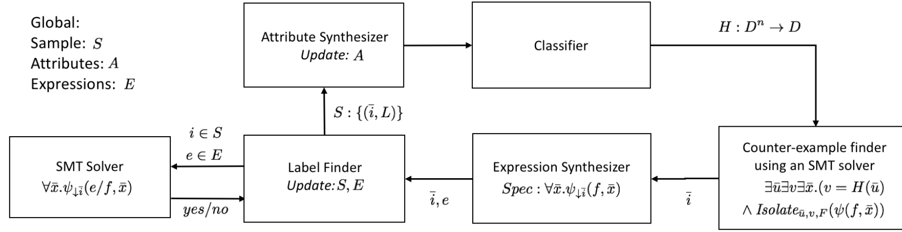


Fig. 1. A General Synthesis Framework based on Learning Classifiers

for $\vec{u}$, and Lemma 3 shows that H is definitely wrong on this input. This will form the basis of generating counterexample inputs in our synthesis framework that we outline in the next section.

3 A General Synthesis Framework by Learning Classifiers

We now present our general framework for synthesizing functions over a first-order theory that uses machine-learning of classifiers. Our technique, as outlined in the introduction, is a *counterexample-guided inductive synthesis approach (CEGIS)*, and works most robustly for single-point refutable specifications.

Given a single-point refutable specification $\forall \vec{x}.\ \psi(f, \vec{x})$, the framework combines several simpler synthesizers and calls to SMT solvers to synthesize a function, as depicted in Figure 1. The solver globally maintains a finite set of expressions E, a finite set of predicates A (also called attributes), and a finite set S of multi-labeled samples, where each sample is of the form $(\vec{p}, Z)$ consisting of an input $\vec{p} \in D^n$ and a set Z of expressions that are correct for $\vec{p}$ (i.e., the specification allows mapping $\vec{p}$ to $e(\vec{p})$ for each $e \in Z$).

Phase 1: In every round, the classifier produces a hypothesis expression H for f. The process starts with a simple expression H, such as one that maps all inputs to a constant. We feed H in every round to a a **counterexample input finder** module, which essentially is a call to an SMT solver to check whether the formula

$$\exists \vec{u}\ \exists v\ \exists \vec{x}.\ (v = H(\vec{u}) \wedge \neg Isolate_{\vec{u},v,F}(\psi(f, \vec{x}))$$

is satisfiable. Note that from the definition of the single-point refutable functions (see Definition 3), whenever H does not satisfy the specification, we are guaranteed that this formula is satisfiable, and the valuation of $\vec{u}$ in the satisfying model gives us an input $\vec{p}$ on which H is definitely wrong (see Lemma 3). If H satisfies the specification, the formula would be unsatisfiable (by Lemma 2) and we can terminate, reporting H as the synthesized expression.
Phase 2: The counterexample input $\vec{p}$ is then fed to an expression synthesizer whose goal is to find *some* correct expression that works for $\vec{p}$. We facilitate this by generating a *new specification for synthesis* that tailors the original specification to the particular input $\vec{p}$. This new specification is the formula

$$\psi{\downarrow_{\vec{p}}}\ (\widehat{f}, \vec{x}) \coloneqq Isolate_{\vec{u},v,T}(\psi(f, \vec{x}))[\vec{p}/\vec{u}, \widehat{f}(\vec{p})/v].$$

The above specification, intuitively, asks for a function $\widehat{f}$ that *"works"* for the input $\vec{p}$. We do this by first finding the formula that isolates the specification to $\vec{u}$ with output v and demand that the specification evaluate to true; then, we substitute $\vec{p}$ for $\vec{u}$ and a new function symbol $\widehat{f}$ evaluated on $\vec{p}$ for v. Any expression synthesized for $\widehat{f}$ in this synthesis problem will map $\vec{p}$ to a value that is consistent with the original specification. We emphasize that we can use any expression synthesizer to synthesize an expression that satisfies this new specification.

Phase 3: Once we synthesized an expression e that works for $\vec{p}$, we feed them to the next phase, which will add e to the set of all expressions E (if e is new), add $\vec{p}$ to the set of samples, and proceed to find the set of *all* expressions in E that work for all the inputs in sample to compute the new set of samples. In order to do this, we take every input $\vec{r}$ that previously existed, and ask whether e works for $\vec{r}$, and if it does, add e to the set of labels for $\vec{r}$. Also, we take the new input $\vec{p}$ and every expression $e' \in E$, and check whether e' works for $\vec{p}$.

To compute this labeling information, we need to be able to check, in general, whether an expression e' works for an input $\vec{r}$. We can do this using a call to an SMT solver that checks whether the formula $\forall \vec{x}\ \psi{\downarrow_{\vec{r}}}\ (e'(\vec{r})/\widehat{f}(\vec{r}),\ \vec{x})$ is valid.

Phase 4: We now have a set of samples, where each sample consists of an input and a set of expressions that work for that input. This is when we look upon the synthesis problem as a *classification* problem— that of mapping every input in the domain to an expression that generalizes the sample (i.e., that maps every input in the sample to *some* expression that it is associated with it). In order to do this, we need to split the input domain into *regions* defined by a class of predicates A. We hence need an *adequate* set of predicates that can define enough regions that can separate the inputs that need to be separated.

Let S be a set of samples and let A be a class of predicates. Two samples $(\vec{j}, E_1)$ and $(\vec{j'}, E_2)$ are said to be *inseparable* if for every predicate $p \in P$, $p(\vec{j}) = p(\vec{j})$. The set of predicates A is said to be adequate for a sample S if any set of inseparable inputs in the sample has a common label as a classification. In other words, for every subset $T \subseteq S$ of the sample set, where $T = \{(\vec{i_1}, E_1), (\vec{i_2}, E_2), \ldots (\vec{i_t}, E_t)\}$, where every pair of inputs in T is inseparable, then $\bigcap_{i=1}^{r} E_i \neq \emptyset$. We require the **attribute synthesizer** to synthesize an adequate set of predicates A, given the set of samples.

Intuitively, if T is a set of pairwise inseparable points with respect to a set of predicates P, then no classifier based on these predicates can separate them, and hence they all need to be classified using the same label; this is possible only if the set of points have a common expression label.

Phase 5: Finally, we give the samples and the predicates to a classification learner, which divides the set of inputs into regions, and maps each region to a single expression, such that the mapping is consistent with the sample. A region is a *conjunction* of predicates and the set of points in the region is the set of all inputs satisfy all these predicates. The classification is consistent with the set of samples if for every sample $(\vec{r}, Z) \in S$, the classifier maps $\vec{r}$ to a label in Z. (In Section 4, we present a general learning algorithm based on decision trees that

learns such a classifier from a set of multi-labeled samples, and which biases the classifier towards small trees.)

The classification synthesized is then converted to an expression in the logic (this will involve nested *ite* expressions using the predicates to define the regions and expressions in leafs to define the function). The synthesized function is fed back to the counterexample input finder, as in Phase 1, and the process continues until we manage to synthesize a function that meets the specification.

4 Multi-Label Decision Tree Classifiers

In this section, we sketch a decision tree learning algorithm for a special case of the so-called multi-label learning problem, which is the problem of learning a predictive model (i.e., a classifier) from samples that are associated with multiple labels. For the purpose of learning the classifier, we assume samples to be vectors of the Boolean values $\mathbb{B} = \{F, T\}$ (inputs returned by the counterexample finder get translated into Boolean values using the global set of attributes). The more general case that datapoints also contain rational numbers can he handled in a straightforward manner as in Quinlan's C4.5 algorithm [20].

To make learning problem precise, let us fix a finite set $L = \{\lambda_1, \ldots, \lambda_k\}$ of labels with $k \geq 2$, and let $\vec{x} = (x_1, \ldots, x_m)$ denote individual samples (in the following also called *datapoints*). The task we are going to solve, which we call *disjoint multi-label learning problem* (cf. Jin and Ghahramani [14]), is

> "given a finite training set $S = \{(\vec{x}_1, Y_1), \ldots, (\vec{x}_n, Y_n)\}$ where $y_i \subseteq L$ and $y_i \neq \emptyset$ for $i \in \{1, \ldots, n\}$, find a decision tree classifier $h \colon \mathbb{B}^m \to L$ such that $h(\vec{x}) \in Y$ for all $(\vec{x}, Y) \in S$."

Note that this learning problem is a special case of the multi-label learning problem studied in machine learning literature, which asks for a classifier that predicts all labels that are associated with a datapoint. Moreover, it is important to emphasize that we require our decision tree classifier to be consistent with the training set (i.e., it is not allowed to misclassify datapoints from training set). In contrast to classical machine learning settings, where classifier are allowed to make (small) errors, we insist on consistency.

We use a straightforward modification of Quinlan's C4.5 algorithm [20] to solve the disjoint multi-label learning problem. (We refer to standard text on machine learning [18] for more information on decision tree learning.) This modification, sketched in pseudocode as Algorithm 1, is a recursive algorithm that constructs a decision tree top-down. More precisely, given a training set S, the algorithm heuristically selects an attribute $i \in \{1, \ldots, m\}$ and splits the set into two disjoint, nonempty subsets $S_i = \{(\vec{x}, Y) \in S \mid x_i = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid x_i = F\}$ (we explain shortly how i is chosen). Then the algorithm recurses on the two subsets, whereby it no longer considers the attribute i. Once the algorithm arrives at a set S' in which all datapoints share a common label (i.e., there exists a $\lambda \in L$ such that $\lambda \in Y$ for all $(\vec{x}, Y) \in S'$), it selects a common label λ (arbitrarily), constructs a single-node tree that is labeled with λ, and returns from the recursion. However, it might happen during construction that a set of datapoints does not

Algorithm 1: Multi-label decision tree learning algorithm

Input: A finite set S of datapoints $x \in \mathbb{B}^m$.

1 **return** `DecTree` $(S, \{1, \dots, m\})$.

2 **Procedure** `DecTree` *(Set of datapoints S, Attributes A)*
3   Create a root node r.
4   **if** *if all datapoints in S have a label in common* **then**
5   Select a common label λ and return the single-node tree r with label λ.
6   **else**
7   Select an attribute $i \in A$ that (heuristically) best splits the sample S.
8   Split S into $S_i = \{(\vec{x}, Y) \in S \mid x_i = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid x_i = F\}$.
9   Label r with attribute i and return the tree with root r, left subtree `DecTree` $(S_i,\ A \setminus \{i\})$, and right subtree `DecTree` $(S_{\neg i},\ A \setminus \{i\})$.
10   **end**

have a common label and cannot be split by any (available) attribute. In this case, it returns an error, as the set of attributes is not adequate (which we make sure does not happen in our framework).

The selection of a "good" attribute to split a a set of datapoints lies at the heart of the decision tree learner as it determines the size of the resulting tree and, hence, how well the tree generalizes the training data. The quality of a split can be formalized by the notion of a *measure*, which, roughly, is a function μ mapping two sets of datapoints to a set R that is equipped with a total order $\preceq$ over elements of R (usually, $R = \mathbb{R}_{\geq 0}$ and $\preceq$ is the natural order over $\mathbb{R}$). Given a set S to split, the learning algorithm first constructs subsets S_i and $S_{\neg i}$ for each available attribute i and evaluates each such candidate split by computing $\mu(S_i, S_{\neg i})$. It then chooses a split that has the least value.

In the single-label setting, information theoretic measures, such as *information gain* (based on *Shannon entropy*) and *Gini*, have proven to produce successful classifiers. In the case of multi-label classifiers, however, finding good measure is still a matter of ongoing research (e.g., see Tsoumakas and Katakis [27] for an overview). However, both the classical entropy and Gini measures can be adapted to the multi-label case in a straightforward way by treating datapoints with multiple labels as multiple identical datapoints with a single label (we describe these in Appendix D). Another modification of entropy has been proposed by Clare and King [7]. However, these approaches share a disadvantage, namely that the association of datapoints to sets of labels is lost and all measures can be high even if all datapoints share a common label; for instance, such a situation occurs for $S = \{(\vec{x}_1, Y_1), \dots, (\vec{x}_n, Y_n)\}$ with $\{\lambda_1, \dots, \lambda_\ell\} \subseteq Y_i$ for $i \in \{1, \dots, n\}$.

Ideally, one would like to have a function that maps to 0 if all datapoints in a set share a common label and to a value strictly greater than 0 if this is not the case. When such a function is used as a measure, a candidate split that produces pure set is guaranteed to have a low value and, hence, is preferred. We now present a measure, based on the combinatorial problem of finding minimal hitting sets, that has this property. To the best of our knowledge, this measure is a novel contribution and has not been studied in the literature.

As *hitting set* of a set S of datapoints we define a set $H \subseteq L$ that satisfies $H \cap Y \neq \emptyset$ for each $(\vec{x}, Y) \in S$. Moreover, we define $hs(S) = \min_{\text{hitting set } H} |H| - 1$ be the cardinality of a smallest hitting set reduced by 1. As desired, we obtain

$hs(S) = 0$ if all datapoints in S share a common label and $hs(S) > 0$ if this is not the case. When evaluating candidate splits, we would prefer to minimize the number of labels needed to label the datapoints in the subsets; however, if two splits agree on this number, we would like to minimize the total number of labels required. Consequently, we propose $R = \mathbb{N} \times \mathbb{N}$ with $(n, m) \preceq (n, m')$ if and only if $n < n'$ or $n = n' \wedge m \leq m'$, and as measures $\mu_{hs}(S_1, S_2) = (\max\{hs(S_1), hs(S_2)\}, hs(S_1) + hs(S_2))$. As computing $hs(S)$ is computationally hard, we implemented a standard greedy algorithm, which runs in time polynomial in the size of the sample.

5 A Synthesis Engine for Linear Integer Arithmetic

We now describe an instantiation of our framework for synthesizing functions expressible in linear integer arithmetic against quantified linear-arithmetic specifications. The *Isolate*() function works over a domain $D \cup \{\bot\}$; we can implement this by choosing a particular element $\widehat{c}$ in the domain and modeling every term using a *pair* of elements, one that denotes the original term and the second that denotes whether the term is $\bot$ or not, depending on whether it is equal to $\widehat{c}$. It is easy to transform the formula now to one that is on the original domain D itself.

We skip a detailed discussion of the counterexample input finder and the label finder because both work essentially by calling an SMT solver. It is not hard to verify that the logical formulas constructed by both components are in linear integer arithmetic if the specification is in linear integer arithmetic. Hence, the implementations of the counterexample input finder and the label finder are straightforward as described in Section 3.

Our implementation globally maintains a finite set E of expressions, a finite set A of predicates, and a finite set S of multi-labeled samples, where each sample is of the form $(\vec{p}, Z)$ consisting of an input $\vec{p} \in D^n$ and a set Z of expressions that are correct for $\vec{p}$. We assume that the sets E and A are initialized with some expressions and attributes, respectively. The exact initialization is unimportant.

Expression Synthesizer Given an input $\vec{p}$, the expression synthesizer has to find an expression that works for $\vec{p}$. Our implementation deviates slightly from the general framework and works in two phases.

In the first phase, it checks whether one of the expressions in the global set E already works for $\vec{p}$. This is done by calling the label finder. If an expression of E already works for $\vec{p}$, the expression synthesizer terminates and we proceed to the label finder. If none of the expressions in E work for $\vec{p}$, the expression synthesizer proceeds to the second phase, where it generates a new synthesis problem with specification $\forall \vec{x} \colon \psi{\downarrow_{\vec{p}}}(\hat{f}, \vec{x})$ according to Phase 2 of Section 3, whose solutions are expressions that work for $\vec{p}$. It solves this synthesis problem using a simple CEGIS-style algorithm, which we sketch next.

Let $\forall \vec{x} \colon \psi(f, \vec{x})$ be a specification with a function symbol $f \colon \mathbb{Z}^n \to Z$, which is to be synthesized, and universally quantified variables $\vec{x} = (x_1, \ldots, x_m)$. Our algorithm synthesizes affine expressions of the form $(\sum_{i=1}^{n} a_i \cdot y_i) + b$ where $y_1, \ldots, y_n$ are integer variables, $a_i \in \mathbb{Z}$ for $i \in \{1, \ldots, n\}$, and $b \in \mathbb{Z}$. The algorithm consists of two components, a *synthesizer* and a *verifier*, which implement

the CEGIS principle in a similar but simpler manner as our general framework. Roughly speaking, the synthesizer maintains an (initially empty) set $V \subseteq \mathbb{Z}^m$ of valuations of the variables $\vec{x}$ and constructs an expression H for the function f that satisfies ψ at least for each valuation in V (as opposed to all possible valuations). Then, it hands this expression over to the verifier. The task of the verifier is to check whether H satisfies the specification. If this is the case, the algorithm has identified a correct expression, returns it, and terminates. Is this not the case, the verifier extracts a particular valuation of the variables $\vec{x}$ for which the specification is violated and hands it over to the synthesizer. The synthesizer adds this valuation to V, and the algorithm iterates.

Both the synthesizer and the verifier reduce their task to a satisfiability problem over linear integer arithmetic and use a constraint solver to solve it. We describe them in detail in Appendix E.

Predicate Synthesizer For the sake of a simpler presentation, we assume that the predicate synthesizer generates a new set of predicate in each iteration from scratch (an implementation would do this incrementally when the attributes are inadequate).

The predicate synthesizer we describe below relies on the observation that the decision tree learning algorithm, which we use as classification leaner component, copes extremely well with a large number of attributes (in fact, the complexity of the learner grows only linearly with the number of attributes). Hence, a straightforward implementation of the predicate synthesizer simply enumerates and adds predicates until it obtains an adequate set.

More precisely, the predicate synthesizer constructs a set A_q of attributes for increasing values of $q \in \mathbb{N}$. The set A_q contains all predicates of the form $\sum_{i=1}^{n} a_i \cdot y_i \leq b$, where y_i are variables corresponding to the function arguments of the function f that is to be synthesized, $a_i \in \mathbb{Z}$ such that each $|a_i| \leq q$, and $|b| \leq q$. Once A_q has been constructed, the predicate synthesizer checks whether A_p is already adequate for S. It can do this by recursively splitting the sample with respect to each predicate in A_q. If all inputs in the resulting subsamples share a common label, the predicate synthesizer has found an adequate set and stops. If it encounters an inseparable set, on the other hand, it increases the parameter q by one and iterates.

Note that the predicate synthesizer is guaranteed to find an adequate set for any sample. The reason for this is that one can separate each input $\vec{p}$ into its own subsample provided q is large enough (larger than absolute value of an entry occurring in the sample).

6 Evaluation

We implemented the framework described in Section 5 for specifications written in the SyGuS format [1]. The implementation is about 5K lines in C++ with API calls to the Z3 SMT solver[8]. The implementations of the LIA counter-example finder and the expression synthesizer use several heuristics. The counter-example finder prioritizes data-points that have a single classification, and returns multiple counterexamples in each round. The expression synthesizer uses a combination of enumeration and constraint-solving, and prioritizes expressions that work for multiple neighboring inputs. More details can be found in Appendix G.

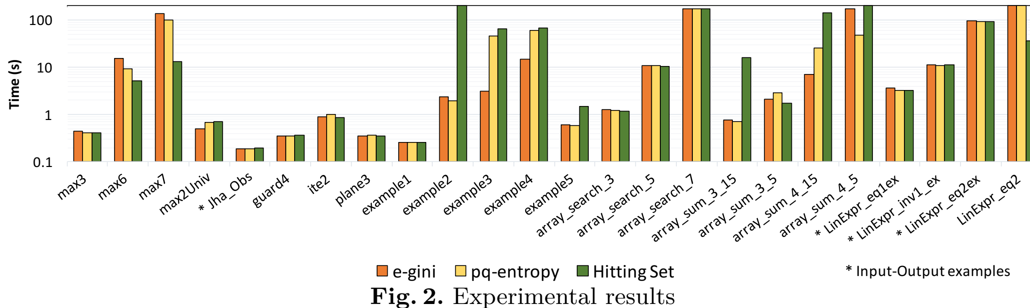


Fig. 2. Experimental results

We evaluated our tool parameterized using the different measures in Section 4 against 43 benchmarks. These benchmarks are predominantly from the 2014 and 2015 SyGuS competition. Additionally, there is an example from [13] for deobfuscating C code using bitwise operations on integers (we query this code 30 times on random inputs, record its output and create an input-output specification for it `Jha_Obs` from it. Moreover, `max2Univ` reformulates the specification for `max2` using universal quantification, as

$\forall x, r, y_1, y_2.\ (r < 0) \Rightarrow ((y_1{=}x \land y_2{=}x{+}r) \lor (y_1{=}x{+}r \land y_2{=}x)) \Rightarrow max_(y_1, y_2) = x$

All experiments were performed on a system with an Intel Core i7-4770HQ 2.20GHz CPU and 4GB RAM running 64-bit Ubuntu 14.04 with a 200 seconds timeout. The results of all the benchmarks are tabulated in Table 1 in Appendix H, while Figure 2 shows results on a representative subset of 26 benchmarks.

Figure 2 compares three measures: *e-gini*, *pq-entropy* and *hitting set* (*pq-entropy* refers to the measure proposed by Clare and King [7]). All solvers time-out on two benchmarks each. None of the algorithms dominate. The hitting-set measure is the only one to solve `LinExpr_eq2`. E-gini and pq-entropy can solve the same set of benchmarks but their performance differs on the `example*` specs, where e-gini performs better, and `max*` where pq-entropy performs better.

The CVC4 SMT-solver based synthesis tool [21] worked very fast on these benchmarks, in general, but it does not *generalize* from underspecifications. On specifications that list a set of input-output examples (marked with $*$ in Figure 2), CVC4 simply returns the precise map that the specification contains, without generalizing it. CVC4 allows restricting the syntax of target functions, but using this feature to force generalization (by disallowing large constants) makes the problems unsolvable for it. CVC4 was also not able to solve the fairly simple specification `max2Ref`.

The general track SyGuS solvers (enumerative, stochastic, constraint-solver, and Sketch) [1] do not work well for these benchmarks (and did not fare well in the competition either); for example, the enumerative solver, which was the winner last year can solve only 15/42 benchmarks.

The above results show that the framework of synthesis developed in this paper that uses theory-specific solvers for basic expressions and combines them using a classification learner yields a competitive solver for the linear-arithmetic domain. We believe more extensive benchmarks are needed to choose the right statistical measures for decision-tree learning.

References

1. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015)
2. Alur, R., D'Antoni, L., Gulwani, S., Kini, D., Viswanathan, M.: Automated grading of DFA constructions. In: IJCAI 2013. IJCAI/AAAI (2013)
3. Alur, R., Singhania, N.: Precise piecewise affine models from input-output data. In: EMSOFT 2014. pp. 3:1–3:10. ACM (2014)
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa (2015), available at `www.SMT-LIB.org`
5. Bemporad, A., Garulli, A., Paoletti, S., Vicino, A.: A bounded-error approach to piecewise affine system identification. IEEE Trans. Automat. Contr. 50(10), 1567–1580 (2005)
6. Cheung, A., Madden, S., Solar-Lezama, A., Arden, O., Myers, A.C.: Using program analysis to improve database applications. IEEE Data Eng. Bull. 37(1), 48–59 (2014)
7. Clare, A., King, R.D.: Knowledge discovery in multi-label phenotype data. In: PKDD 2001. LNCS, vol. 2168, pp. 42–53. Springer (2001)
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
9. Ferrari-Trecate, G., Muselli, M., Liberati, D., Morari, M.: A clustering technique for the identification of piecewise affine systems. Automatica 39(2), 205–217 (2003)
10. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer (2014)
11. Garg, P., Madhusudan, P., Neider, D., Roth, D.: Learning invariants using decision trees and implication counterexamples. POPL 2016 p. to appear (2016)
12. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL 2011. pp. 317–330. ACM (2011)
13. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 215–224. ICSE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806799.1806833
14. Jin, R., Ghahramani, Z.: Learning with multiple labels. In: NIPS 2002. pp. 897–904. MIT Press (2002)
15. Karaivanov, S., Raychev, V., Vechev, M.T.: Phrase-based statistical translation of programming languages. In: Onward!, part of SLASH '14. pp. 173–184. ACM (2014)
16. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: AAIP 2009, Revised Papers. LNCS, vol. 5812, pp. 50–73. Springer (2010)
17. McClurg, J., Hojjat, H., Cerný, P., Foster, N.: Efficient synthesis of network updates. In: PLDI 2015. pp. 196–207. ACM (2015)
18. Mitchell, T.M.: Machine learning. McGraw Hill series in computer science, McGraw-Hill (1997)

19. Paoletti, S., Juloski, A.L., Ferrari-Trecate, G., Vidal, R.: Identification of hybrid systems: A tutorial. Eur. J. Control 13(2-3), 242–260 (2007)
20. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
21. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer (2015)
22. Saha, S., Garg, P., Madhusudan, P.: Alchemist: Learning guarded affine functions. In: CAV 2015. LNCS, vol. 9206, pp. 440–446. Springer (2015)
23. Saha, S., Prabhu, S., Madhusudan, P.: Netgen: Synthesizing data-plane configurations for network policies. In: SOSR 2015. pp. 17:1–17:6. ACM (2015)
24. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: PLDI 2013. pp. 15–26. ACM (2013)
25. Solar-Lezama, A.: Program sketching. STTT 15(5-6), 475–495 (2013)
26. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS 2006. pp. 404–415. ACM (2006)
27. Tsoumakas, G., Katakis, I.: Multi-label classification: An overview. IJDWM 3(3), 1–13 (2007)
28. Vidal, R., Soatto, S., Ma, Y., Sastry, S.: An algebraic geometric approach to the identification of a class of linear hybrid systems. In: Decision and Control, 2003. Proceedings. 42nd IEEE Conference on. vol. 1, pp. 167–172 (Dec 2003)

A Proof of Lemma 2

Proof. By induction, we can show that for any interpretation of $\vec{x}$ and $\vec{u}$ over the domain D, and with $v = h(u)$, whenever a term $Isol_{\vec{u},v,F}(t)$ evaluates to a non-$\bot$ value (i.e., to a value in D), t will evaluate to the same value, and whenever a formula $Isol_{\vec{u},v,F}(\varphi)$ evaluates to false, the formula φ will evaluate to false as well. □

B Proof of Lemma 3

Proof. If h is a function that satisfies the specification and mapped $\vec{p}$ to q, then for any interpretation of $\vec{x}$, $\psi(f,\ \vec{x}\,)$ evaluates to true. We can show that $Isolate_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ will either evaluate to true or to $\bot$, since it essentially interprets f on $\vec{p}$ to q and the rest to $\bot$, and checks whether ψ would evaluate to false independent of how f maps other inputs. □

C Proof of Lemma 4

Proof. Let $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ be a single-point refutable specification, and assume that it is not single-point definable. Let $\mathcal{F}$ be the class of all functions h that satisfy this specification. Then there is some function $h' : D^n \rightarrow D$ such that for every input $\vec{p}$, there exists some function $h \in \mathcal{F}$ such that $h'(\vec{p}) = h(\vec{p})$, and yet h' does not satisfy the specification. By single-point refutability of the specification, there must be some input $\vec{p}$ such that when we interpret $v = h'(\vec{p})$, there is an interpretation of $\vec{x}$ such that $Isolate_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ evaluates to false. Let $h \in \mathcal{F}$ be some function that agrees with h' on $\vec{p}$. By Lemma 3, there is no function that satisfies the specification and that maps $\vec{u}$ to v, which contradicts the fact that h' satisfies the specification. □

D Measure for Learning Decision Tree Classifiers

In the following, we describe how to adapt entropy and Gini measures to the multi-label case; the main idea is to treat a single datapoint $(\vec{x}, \{\lambda_1, \ldots, \lambda_k\})$ with the datapoints $(\vec{x}, \lambda_1), \ldots, (\vec{x}, \lambda_1)$ and to proceed as in classical decision tree learning. Moreover, we briefly sketch the modified entropy measure proposed by Clare and King [7]. In all cases, we have $R = \mathbb{R}$ and $\preceq$ is the natural order over $\mathbb{R}$.

Entropy Intuitively, entropy is a measure for the amount of "information" contained in a sample; the higher the entropy, the higher the randomness of the sample. Formally, one defines the entropy of a sample S with multiple labels by $e(S) = -\sum_{\lambda \in L} p_\lambda \cdot \log_2 p_\lambda$ where p_λ is the relative frequency of the label λ (i.e., $p_\lambda = |\{(x, Y) \in S \mid \lambda \in Y\}| \,/\, \sum_{(x,Y) \in S} |Y|$). The corresponding measure $\mu_e(S_1, S_2)$ is the weighted average of $e(S_1)$ and $e(S_2)$.

Gini One can think of Gini as the probability of making a classification error if the whole sample is uniformly labeled with a randomly chosen label. Formally, for a sample S with multiple labels, one defines Gini by $g(S) = \sum_{\lambda \neq \lambda' \in L} p_\lambda \cdot p_{\lambda'}$ where p_λ is as above. The Gini measure $\mu_g(S_1, S_2)$ is the weighted average of $g(S_1)$ and $g(S_2)$.

pq-entropy Clare and King's [7] modification of entropy accounts for multiple labels by considering for each label the probability of being labeled with λ (i.e., the relative frequency of λ) as well as probability of not being labeled with λ. More precisely, for a sample S with multiple labels, Clare and King define $pq\text{-}e = -\sum_{\lambda \in L} p_\lambda \cdot \log_2 p_\lambda + q_\lambda \cdot \log_2 q_\lambda$ where p_λ is as above and $p_\lambda = 1 - p_\lambda$. As measure $\mu_{pq\text{-}e}(S_1, S_2)$, Clare and King use the weighted average of $pq\text{-}e(S_1)$ and $pq\text{-}e(S_2)$.

E Details of the Expression Synthesizer

Synthesizer The synthesizer maintains a finite set $V \subseteq \mathbb{Z}^m$ of valuations of the universally quantified variables $\vec{x}$ and constructs expressions for the synthesis function f that satisfies ψ at least on all valuations in V. To this end, the synthesizer first constructs a template expression $t(\vec{a}, b, \vec{y})$ of the form described above, but where $\vec{a} = (a_1, \ldots, a_n)$, and b are now variables (note that this expression is not linear due to the terms $a_i \cdot y_i$). Then, it constructs the formula

$$\varphi(\vec{a}, b) \coloneqq \bigwedge_{\vec{v} \in V} \psi(t/f, \vec{v}/\vec{x}).$$

Note that φ is a formula in linear integer arithmetic since all occurrences of variables y_i have been replaced with integers values. Finally, the algorithm uses an SMT solver to obtain valuations of $\vec{a}$ and b that satisfy φ; note that φ is guaranteed to be satisfiable since we use the synthesizer in a special setting, namely to synthesize expressions for a single-point definable specification. The synthesizer substitutes the satisfying assignment for $\vec{a}$ and b in the template t and returns the resulting expression H.

Verifier Given an expression H conjectured by the synthesizer, the verifier has to check whether

$$\varphi \coloneqq \psi(H/f, \vec{x})$$

is valid. To this end, the verifier turns this validation problem into a satisfiability problem by querying an SMT solver whether $\neg\varphi$ is satisfiable. If $\neg\varphi$ is satisfiable, then the verifier extracts a satisfying assignment $\vec{v}$ for the universal quantified variables and returns $\vec{v}$ to the synthesizer. If $\neg\varphi$ is unsatisfiable, an expression satisfying the specification has been found and the synthesizing algorithm returns it.

F Details of the Expression Synthesizer

Classification Learner We use the decision tree learner described in Section 4 to learn a decision tree classifier over the samples S and the predicates A. In a preparatory step, the classification learner transforms each input $\vec{p} \in S$ to a Boolean vector $\vec{b}_{\vec{p}}$: given $\vec{p}$ and predicates $A = \{p_1, \ldots, p_m\}$, it constructs the Boolean vector $\vec{b}_{\vec{p}} = (p_1(\vec{p}), \ldots, p_m(\vec{p})) \in \mathbb{B}^m$. It collects all transformed inputs in a new sample S', where the label of $\vec{b}_{\vec{p}}$ is the classification of $\vec{p}$. (Also here, one would clearly construct S' incrementally, growing it in each iteration.)

Once the set S' has been created, we run the decision tree learner on S'. The result is a tree τ, say with root node v_r, whose inner nodes are labeled with predicates from A and whose leafs are labeled with expression from E. The formula in linear integer arithmetic corresponding to τ is the nested if-then-else expression $\textit{to-ite}(v_r)$, where $\textit{to-ite}(v)$ for a tree node v is recursively defined by

- if v is a leaf node labeled with expression e, then $\textit{to-ite}(v) \coloneqq e$; and
- if v is an inner node labeled with predicate p and children v_1 and v_2, then $\textit{to-ite}(v) \coloneqq \textit{ite}(p, \textit{to-ite}(v_1), \textit{to-ite}(v_2))$.

The classification learner finally returns $\textit{to-ite}(v_r)$.

G Implementation details of the LIA solver

The technical aspects of the Counter-example finder and the Label Finder, is implemented a bit-differently than explained earlier in 5. We use array theories and uninterpreted functions to extract the counterexample point from the specification and the hypothesis, and to check if a point satisfies an expression. To aid the learner, we generate up-to five counterexample points in each iteration of the learning loop which helps us converge faster.

We maintain an initial set of enumerated expressions, and dovetail through them lazily before invoking the expression synthesizer. These initial expressions has coefficients between -1 and 1 for all the variables. Once an counter-example point does not satisfy any of these enumerated expressions, the expression synthesizer is invoked.

The LIA expression synthesizer is based on the following observation and heuristic from standard geometric approaches. To synthesize a d-dimension plane, we need at-least (d+1) points which lies on that plane. And if $< y_1, y_2, \ldots y_d >$ is a point, then it is highly likely that the expression that satisfies it, would also pass through its immediate neighboring points in each dimension, namely $< y_1 + 1, y_2 \ldots y_d >, < y_1, y_2 + 1 \ldots y_d >, \ldots < y_1, y_2, \ldots y_d + 1 >$. We constraint the SMT solver to synthesize a d-dimensional expression with integer coefficient, that satisfies these (d+1) points.

H Experimental Results

Benchmarks	e-gini		pq-entropy		Hitting Set	
	#rounds	Time(s)	#rounds	Time(s)	#rounds	Time(s)
Jha_Obs.sl	1	0.19	1	0.19	1	0.2
LinExpr_eq1.sl	–	TO	–	TO	–	TO
LinExpr_eq1ex.sl	9	3.64	10	3.3	10	3.22
LinExpr_eq2.sl	–	TO	–	TO	31	36.84
LinExpr_eq2ex.sl	48	94.53	49	93.62	50	93.14
LinExpr_inv1_ex.sl	39	11.49	39	10.88	38	11.1
max2Ref.sl	9	0.5	9	0.69	10	0.72
fg_array_search_2.sl	7	0.64	7	0.64	6	0.57
fg_array_search_3.sl	7	1.28	7	1.22	7	1.19
fg_array_search_4.sl	12	4.46	11	4.37	16	4.55
fg_array_search_5.sl	19	10.69	20	10.7	16	10.57
fg_array_search_6.sl	22	50.28	22	49.96	22	49.34
fg_array_search_7.sl	27	174.61	22	174.05	20	170.1
fg_array_sum_2_15.sl	5	0.32	5	0.31	5	0.31
fg_array_sum_2_5.sl	7	0.39	7	0.38	12	0.61
fg_array_sum_3_15.sl	9	0.77	9	0.71	40	15.83
fg_array_sum_3_5.sl	31	2.11	39	2.87	20	1.72
fg_array_sum_4_15.sl	28	7.09	76	26.09	44	139.53
fg_array_sum_4_5.sl	187	169.77	105	47.22	–	TO
fg_max2.sl	3	0.21	3	0.21	3	0.21
fg_max3.sl	8	0.44	8	0.42	8	0.42
fg_max4.sl	18	1	16	0.87	12	0.69
fg_max5.sl	44	2.8	51	3.35	32	2.22
fg_max6.sl	130	15.42	94	9.28	47	5.19
fg_max7.sl	327	136.18	271	98.58	65	13.24
fg_max8.sl	–	TO	–	TO	140	92.04
fg_mpg_example1.sl	3	0.26	3	0.26	3	0.26
fg_mpg_example2.sl	31	2.42	30	1.98	–	TO
fg_mpg_example3.sl	10	3.17	12	46.75	14	65.33
fg_mpg_example4.sl	29	15.05	46	61.11	52	66.73
fg_mpg_example5.sl	9	0.6	9	0.58	20	1.47
fg_mpg_guard1.sl	3	0.3	3	0.31	3	0.31
fg_mpg_guard2.sl	2	0.27	2	0.25	2	0.25
fg_mpg_guard3.sl	4	0.37	4	0.35	4	0.36
fg_mpg_guard4.sl	4	0.35	4	0.36	4	0.37
fg_mpg_ite1.sl	4	0.68	4	0.65	4	0.67
fg_mpg_ite2.sl	9	0.89	11	1.01	7	0.86
fg_mpg_plane1.sl	1	0.19	1	0.19	1	0.2
fg_mpg_plane2.sl	1	0.37	1	0.36	1	0.38
fg_mpg_plane3.sl	1	0.36	1	0.37	1	0.36
s1.sl	5	0.26	5	0.27	5	0.26
s2.sl	2	0.22	2	0.22	2	0.22
s3.sl	1	0.19	1	0.2	1	0.19

Table 1. Experimental Results