# Compositional Synthesis of Piece-Wise Functions by Learning Classifiers

DANIEL NEIDER, RWTH Aachen University
SHAMBWADITYA SAHA and P. MADHUSUDAN, University of Illinois at Urbana-Champaign

We present a novel general technique that uses classifier learning to synthesize piece-wise functions (functions that split the domain into regions and apply simpler functions to each region) against logical synthesis specifications. Our framework works by combining a synthesizer of functions for fixed 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 integer arithmetic, combining leaf expression synthesis using constraint-solving with predicate synthesis using enumeration, and tie them together using a decision tree classifier. We demonstrate that this compositional approach is competitive compared to existing synthesis engines on a set of synthesis specifications.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; Constraint and logic languages; • **Computing methodologies** → *Supervised learning by classification*; • **Theory of computation** → Computational geometry;

Additional Key Words and Phrases: Program synthesis, piece-wise functions, constraint solving, machine learning, counterexamples

**ACM Reference format:**
Daniel Neider, Shambwaditya Saha, and P. Madhusudan. 2018. Compositional Synthesis of Piece-Wise Functions by Learning Classifiers. *ACM Trans. Comput. Logic* 19, 2, Article 10 (May 2018), 23 pages.
https://doi.org/10.1145/3173545

## 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 (Alur et al. 2015) to automatically synthesize small expressions or programs that are useful in niche application domains, including end-user programming (Gulwani 2011), filling holes in program sketches (Solar-Lezama et al. 2006), program transformations (Karaivanov et al. 2014; Cheung et al. 2014), automatic grading of assignments (Alur et al. 2013; Singh et al. 2013), synthesizing network configurations and migrations (Saha et al. 2015b; McClurg et al. 2015), as well as synthesizing annotations, such as invariants or pre/post conditions for programs (Garg et al. 2014a, 2016).

A shorter version of this article was presented at the conference TACAS (Neider et al. 2016). This material is based upon work supported by the National Science Foundation under Grants No. 1138994 and No. 1527395.

Authors' addresses: D. Neider, Chair of Computer Science 7, RWTH Aachen University, 52074 Aachen, Germany; email: neider@automata.rwth-aachen.de; S. Saha and P. Madhusudan, University of Illinois at Urbana-Champaign, Department of Computer Science, 201 North Goodwin Avenue, Urbana, IL 61801-2302, USA; emails: {ssaha6, madhu}@illinois.edu.

https://doi.org/10.1145/3173545

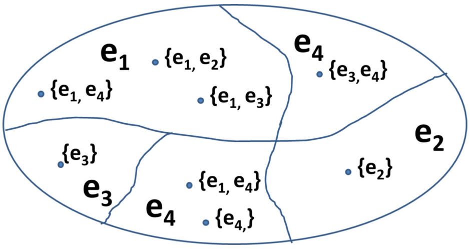

Fig. 1. Learning classifiers.

The field of machine learning (Mitchell 1997) 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 (Kitzelmann 2010). Machine learning, which is the field of learning algorithms that can build predictors on data using samples/training data, is a rich field that encompasses algorithms for several problems, including classification, regression, and clustering (Mitchell 1997).

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 (Solar-Lezama et al. 2006). A majority of the current synthesis approaches rely on counterexample guided inductive synthesis (Solar-Lezama et al. 2006; Garg et al. 2014a, 2016; Jha et al. 2010).

In this article, 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 $\forall \vec{x}.\ \psi(\vec{x})$.[1] We will assume that $\psi$ 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 produce models when a formula is satisfiable. The goal of this article is to develop a 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 Figures 1 and 2 on Page 11):

- 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, then 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. The notion of when an expression

[1] Note that this syntax is expressive enough, of course, to describe input-output examples as well.

maps an input to a correct value is also an unclear concept, which we will discuss later. The 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 sets of 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 that have disjoint sets of labels. Once we have such a set of predicates, we are equipped with an adequate granularity of regions to find a piece-wise function.
- The final phase uses *classification learning* to generalize the samples to a function from all inputs to basic expressions (see figure above). The learning should be biased toward finding *simple* functions, finding few 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. First, 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. However, it turns out that, in general, there may be *no* single input point that can demonstrate a violation of the specification (e.g., the specification $\psi(f, x) := f(x+1) \neq f(x)$ requires two input points, namely a concrete value for $x$ and $x + 1$, to show that a hypothesis is incorrect), and even if there was, finding one may be hard. To address this problem, we

- develop a theory of *single-point definable specifications* whose definition ensures that single counterexample inputs always exist; and
- define a subclass of such specifications, called *single-point refutable specifications*, for which finding such counterexample inputs can be reduced to satisfiability problems over the underlying logical domain (which is assumed to be decidable), hence providing an effective way to compute counterexample inputs.

Our framework works robustly for the class of single-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 (Solar-Lezama et al. 2006; Alur et al. 2015), 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 (since the actual arguments of $f$ can be arbitrary complex expression over $\vec{x}$), and hence cannot be used in our framework. A simple example where this problem manifests is the specification $\psi(f, x) := f(x+1) = 0$; given the hypothesis $H$ with $H(1) = 1$ and $H(x) = 0$ for all $x \neq 1$, a model for $\neg\psi(H/f, x)$ would assign $x$ the value 0, though 1 is the actual input on which $H$ is incorrect. By contrast, the class of counterexamples we use is strictly presented as inputs for the function being synthesized, and is different from those used in current

CEGIS approaches. The framework of single-point refutable specifications and the counterexample input generation procedures we build for them is crucial to be able to use classifiers to synthesize expressions.

The classifier learning algorithm can be any learning algorithm for multi-label classification (preferably 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 is noisy. In Section 4, we describe the second contribution of this article—an adaptation of decision-tree learning to multi-label learning that produces classifiers that are consistent with the sample. (We refer the reader to Mitchell (1997) and Quinlan (1993) for further details about decision tree learning.) We also explore a variety of statistical measures used within the decision-tree learning algorithm to bias the learning towards smaller trees in the presence of multi-labeled samples. The resulting decision-tree learning algorithms form one class of classifier learning algorithms that can be used to synthesize piece-wise functions over any theory that works using our framework.

The third contribution of the article is an instantiation of our framework to build an efficient synthesizer of piece-wise linear integer arithmetic functions for specifications given in the theory of linear integer 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 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 using a constraint solver. The predicate synthesizer is instantiated using an enumerative synthesis algorithm. The resulting solver works extremely well on a large class of benchmarks drawn from the SyGuS 2015 synthesis competition (Alur et al. 2016a) (linear integer arithmetic 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 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) (Alur et al. 2015), which provides a language, similar to SMTLib (Barrett et al. 2015), to describe synthesis problems. Several solvers following the counterexample-guided inductive synthesis approach (CEGIS) (Solar-Lezama et al. 2006) for SyGuS have been developed (Alur et al. 2015), including an enumerative solver, a solver based on constraint solving, one based on stochastic search, and one based on the program synthesizer Sketch (Solar-Lezama 2013). Recently, a solver based on CVC4 (Reynolds et al. 2015) has also been presented.

There has been several works on synthesizing piece-wise affine models of hybrid dynamical systems from input-output examples (Alur and Singhania 2014; Bemporad et al. 2005; Ferrari-Trecate et al. 2003; Vidal et al. 2003) (we refer the reader to Paoletti et al. (2007) 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 (Saha et al. 2015a). In contrast to our setting, it requires that the function to synthesize is unique.

The learning framework we develop in this article, as well as the synthesis algorithms we use for linear-arithmetic (the outer learner, the expression synthesizer and the predicate synthesizer) can be seen as an abstract learning framework (Löding et al. 2016).

## 2 THE SYNTHESIS PROBLEM AND SINGLE-POINT REFUTABLE SPECIFICATIONS

The synthesis problem we tackle in this article is that of finding a function $f$ that satisfies a logical specification of the form $\forall \vec{x}.\ \psi(f, \vec{x})$, where $\psi$ is a *quantifier-free* first-order formula over a logic with fixed interpretations of constants, functions, and relations (except for $f$). 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 to generate counterexamples while performing synthesis.

For the rest of the article, let $f$ be a function symbol with arity $n$ representing the target function that is to be synthesized. The specification logic is a formula in first-order logic, over an arbitrary set of function symbols $\mathcal{F}$, (including a special symbol $f$), constants $\mathcal{C}$, and relations/predicates $\mathcal{P}$, all of which 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 assume that negation is pushed into atomic predicates.

The specification for synthesis is a formula of the form $\forall \vec{x}.\ \psi(f, \vec{x})$, where $\psi$ is a formula expressed in the following grammar (where $g \in \mathcal{F}$, $c \in \mathcal{C}$, and $P \in \mathcal{P}$):

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

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

The synthesis problem is to find, given a specification $\forall \vec{x}.\ \psi(f, \vec{x})$, a definition for the function $f$ in a particular syntax that satisfies the specification. More formally, given a subset of function symbols $\widehat{\mathcal{F}} \subseteq \mathcal{F}$ (excluding $f$) and a subset of constants $\widehat{\mathcal{C}}$ and a subset of relation/predicate symbols $\widehat{\mathcal{P}} \subseteq \mathcal{P}$, the task is to find an *expression* $e$ for $f$ that is a term with free variables $y_1, \ldots, y_n$ adhering to the following syntax (where $\widehat{g} \in \widehat{\mathcal{F}}$, $\widehat{c} \in \widehat{\mathcal{C}}$, $\widehat{P} \in \widehat{\mathcal{P}}$):

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

such that $e$ satisfies the specification (i.e., $\forall \vec{x}.\ \psi(e/f,\ \vec{x})$ is valid).

### 2.1 Single-Point Definable Specifications

To be able to define a general 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 does not satisfy the specification with a concrete input on which $H$ 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 allows 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 input. 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 6 for several examples taken from a recent 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 2.1 (Single-Point Definable (SPD) Specifications).* A specification $\alpha$ is said to be *single-point definable* if the following holds. Let $\mathcal{F}$ be the class of all functions that satisfy the specification $\alpha$. 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 $\alpha$).

Intuitively, a specification is single-point definable if whenever we construct a function that maps each input independently according to *some* arbitrary function that satisfies the specification, the resulting function satisfies the specification as well. For each input $\vec{p}$, if $X_{\vec{p}}$ is the set of all outputs that functions 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 of each input independent of other inputs.

Let us illustrate this definition with the following examples.

*Example 2.2.* Consider the following specifications in the first-order theory of arithmetic:

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

While single-point definable specifications are quite common, there are prominent specifications that are not single-point definable. For example, *inductive loop invariant synthesis* specifications for programs are not single-point definable, as counterexamples to the inductiveness constraint involve *two counterexample inputs* (the ICE learning model (Garg et al. 2014a) formalizes this). Similarly, ranking function synthesis is also not single-point definable.

Note that for any single-point definable specification, 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 directly from the definition.

Proposition 2.3. *Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a single-point definable specification and let $h : D^n \to D$ be an interpretation for $f$ such that $\forall \vec{x}.\ \psi(f, \vec{x})$ does not hold. Then, there exists an input $\vec{p} \in D^n$ such that for* every *function $h' : D^n \to D$ that satisfies the specification, $h(\vec{p}) \neq h'(\vec{p})$.*

## 2.2 Single-Point Refutable Specifications

While the above proposition ensures that there is a counterexample input for any hypothesized function that does not satisfy a single-point definable function, it does not ensure that finding such an input is tractable. We now define single-point refutable specifications, which we show to be a subclass of single-point definable specifications, and for which we can reduce the problem of finding counterexample inputs 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 true, and where the truth-hood is caused *solely* by the interpretation of $H$ on $\vec{p}$. 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.

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, 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 $\vec{u}$ 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 $D \cup \{\bot\}$, where $\bot$ 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 $\vec{u}$. We define this alternate semantics by *transforming* a formula $\psi(f, \vec{x})$ to a formula with the usual semantics but over an extended domain $D^+ = D \cup \{\bot\}$. In this transformation, we use if-then-else (*ite*) terms for simplicity. Moreover, given a vector $\vec{z} = (z_1, \ldots, z_\ell)$ (e.g., of variables), we use $\vec{z}[i]$ as a shorthand for the $i$-th entry $z_i$ of $\vec{z}$ (i.e., $z[i] = z_i$) throughout the rest of this article.

*Definition 2.4 (Isolate Transformer).* Let $\vec{u}$ be a vector of $n$ first-order variables (where $n$ is the arity of the function to be synthesized), $v$ a first-order variable (different from ones in $\vec{u}$), and $b \in \{T, F\}$ a Boolean value. Moreover, 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 $D^+$ by

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

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

$$\begin{aligned}
Isol_{\vec{u},v,b}(x) &:= x,\\
Isol_{\vec{u},v,b}(c) &:= c,\\
Isol_{\vec{u},v,b}(g(t_1,\ldots,t_k)) &:= ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \bot, g\Big(Isol_{\vec{u},v,b}(t_1),\ldots,Isol_{\vec{u},v,b}(t_k)\Big)\right),\\
Isol_{\vec{u},v,b}(f(t_1,\ldots,t_n)) &:= ite\left(\bigwedge_{i=1}^{n} Isol_{\vec{u},v,b}(t_i) = \vec{u}[i], v, \bot\right),\\
Isol_{\vec{u},v,b}(P(t_1,\ldots,t_k)) &:= ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \neg b, P\Big(Isol_{\vec{u},v,b}(t_1),\ldots,Isol_{\vec{u},v,b}(t_k)\Big)\right),\\
Isol_{\vec{u},v,b}(\neg P(t_1,\ldots,t_k)) &:= ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \neg b, \neg P\Big(Isol_{\vec{u},v,b}(t_1),\ldots,Isol_{\vec{u},v,b}(t_k)\Big)\right),\\
Isol_{\vec{u},v,b}(\varphi_1 \vee \varphi_2) &:= Isol_{\vec{u},v,b}(\varphi_1) \vee Isol_{\vec{u},v,b}(\varphi_2),\\
Isol_{\vec{u},v,b}(\varphi_1 \wedge \varphi_2) &:= Isol_{\vec{u},v,b}(\varphi_1) \wedge Isol_{\vec{u},v,b}(\varphi_2).
\end{aligned}$$

Intuitively, the function $\mathit{Isolate}_{\vec{u},v,b}(\psi)$ captures whether $\psi$ 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 $\perp$. Functions on terms that involve $\perp$ are sent to $\perp$ as well. Predicates are evaluated to $b$ only if the predicate is evaluated on terms none of which is $\perp$—otherwise, they get mapped to $\neg b$, to reflect that it will not help to make the final formula $\psi$ evaluate to $b$. Note that when $\mathit{Isolate}_{\vec{u},v,b}(\psi)$ evaluates to $\neg b$, there is no property of $\psi$ that we claim. Also, note that $\mathit{Isolate}_{\vec{u},v,b}(\psi(f,\vec{x}))$ has no occurrence of $f$ in it, but has free variables $\vec{x}$, $\vec{u}$ and $v$. The following examples illustrates the isolate transformer.

*Example 2.5.* Consider the (single-point refutable) specification

$$\psi(f,x) = f(x) > x + 1$$

in linear integer arithmetic over a single variable $x$. The formula $\mathit{Isol}_{\vec{u},v,b}$ will have free variables $x$, $u$, and $v$ (note that $x$ and $u$ are variables not vectors of variables in this example).

In the first step, we adapt the semantics of the operator $+$ and the predicate $>$ to account for the new value $\perp$ by introducing a new operator $+_\perp$ and a new predicate $>_\perp$. Given two terms $t_1$ and $t_2$, the operator $+_\perp$ is defined by

$$t_1 +_\perp t_2 := \mathit{ite}\Big(t_1 = \perp \vee t_2 = \perp, \perp, t_1 + t_2\Big),$$

while the predicate $>_\perp$ is defined by

$$t_1 >_\perp t_2 := \mathit{ite}\Big(t_1 = \perp \vee t_2 = \perp, \perp, t_1 > t_2\Big).$$

In both cases, the result is $\perp$ if one one of the terms evaluates to $\perp$, whereas the original semantics is retained otherwise.

In the second step, we can now apply the isolate transformer to $\psi$:

$$\begin{aligned} \mathit{Isol}_{\vec{u},v,b}(\psi(f,x)) &= \mathit{Isol}_{\vec{u},v,b}(f(x) > x + 1) \\ &= \mathit{Isol}_{\vec{u},v,b}(f(x)) >_\perp \Big(\mathit{Isol}_{\vec{u},v,b}(x) +_\perp \mathit{Isol}_{\vec{u},v,b}(1)\Big) \\ &= \mathit{ite}(x = u, v, \perp) >_\perp (x +_\perp 1). \end{aligned}$$

In total, we obtain

$$\mathit{Isolate}_{\vec{u},v,b}(\psi(f,x)) = \mathit{ite}\Big(x = \perp, \neg b, \mathit{ite}(x = u, v, \perp) >_\perp (x +_\perp 1)\Big),$$

which captures whether $\psi$ will evaluate to $b$ if $f$ maps $u$ to $v$ (and independent of how $f$ is interpreted on other inputs).

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. This is formalized below.

Lemma 2.6. *Let $\forall\vec{x}.\ \psi(f,\ \vec{x})$ be a specification and $h : D^n \to D$ a function satisfying the specification. Then, there is no interpretation of the variables in $\vec{u}$ and $\vec{x}$ (over $D$) such that if $v$ is interpreted as $h(\vec{u})$, the formula $\mathit{Isolate}_{\vec{u},v,F}(\psi(f,\ \vec{x}))$ evaluates to false.*

Proof. Let $\forall\vec{x}.\ \psi(f,\vec{x})$ be a specification and $h : D^n \to D$ a function satisfying the specification. Moreover, let $\vec{u}$ a vector of variables over the domain $D$, $v$ a variable over $D$, and $b \in \{T, F\}$ a Boolean value. Finally, fix a valuation $d_z \in D$ for each free variable $z$ in $\mathit{Isolate}_{\vec{u},v,b}(\psi(f,\vec{x}))$ such that $d_v = h(d_{u[1]}, \ldots, d_{u[n]})$.

We split the proof into two parts:

(1) We show that if $\mathit{Isol}_{\vec{u},v,b}(t)$ evaluates to a non-$\perp$ value (i.e., to a value in $D$) for a term $t$, then $t$ evaluates to the same value.

(2) Using Part 1, we show that if $Isol_{\vec{u},v,F}(\varphi)$ evaluates to false for a formula $\varphi$, the formula $\varphi$ evaluates to false as well.

The claim of Lemma 2.6 then follow immediately from Part 2 and the definition of $Isolate_{\vec{u},v,b}$, since $h$ satisfies the specification and the variable $v$ is interpreted as $h(\vec{u})$.

*Proof of Part 1.* We prove the first part using an induction over the structure of a term $t$.

*Base case.* Let $t = x$ or $t = c$. Then, the claim holds immediately by definition of $Isol_{\vec{u},v,b}$.
*Induction step.* In the induction step, we distinguish between $t = g(t_1, \ldots, t_k)$ and $t = f(t_1, \ldots, t_n)$.

- Let $t = g(t_1, \ldots, t_k)$ and assume that $Isol_{\vec{u},v,b}(t)$ evaluates to a non-$\bot$ value, say $d \in D$. By definition of $Isol_{\vec{u},v,b}$, this means that $Isol_{\vec{u},v,b}(t_i)$ evaluates to a non-$\bot$ value, say $d_i \in D$, for each $i \in \{1, \ldots, k\}$. Moreover, $Isol_{\vec{u},v,b}(t)$ evaluates to $g(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k))$ and, hence, $d = g(d_1, \ldots, d_k)$. Applying the induction hypothesis now yields that $t_i$ also evaluates to $d_i$. Since $t = g(t_1, \ldots, t_k)$, this means that $t$ evaluates to $d$, as claimed.
- Let $t = f(t_1, \ldots, t_n)$ and assume that $Isol_{\vec{u},v,b}(t)$ evaluates to a non-$\bot$ value. By definition of $Isol_{\vec{u},v,b}$, this means that $Isol_{\vec{u},v,b}(t_i) = \vec{u}[i]$ for $i \in \{1, \ldots, n\}$, Moreover, $Isol_{\vec{u},v,b}(t)$ evaluates to $d_v$. Applying the induction hypothesis now yields that $t_i$ evaluates to $d_{\vec{u}[i]} \in D$ for each $i \in \{1, \ldots, n\}$. Since $t = f(t_1, \ldots, t_n) = f(\vec{u}[1], \ldots, \vec{u}[n])$ and $v$ is interpreted as $h(\vec{v})$, this means that $t$ evaluates to $h(d_{\vec{u}[1]}, \ldots, d_{\vec{u}[n]}) = d_v$, as claimed.

*Proof of Part 2.* We prove the second part using an induction over the structure of a formula $\varphi$. Recall that we fix $b = F$ for this part of the proof.

*Base case.* In the induction step, we distinguish between the two cases $\varphi = P(t_1, \ldots, t_k)$ and $\varphi = \neg P(t_1, \ldots, t_k)$.

- Let $\varphi = P(t_1, \ldots, t_k)$ and assume that $Isol_{\vec{u},v,F}(\varphi)$ evaluates to false. By definition of $Isol_{\vec{u},v,b}$, this means that $Isol_{\vec{u},v,F}(t_i)$ evaluates to a non-$\bot$ value, say $d_i \in D$, for each $i \in \{1, \ldots, k\}$. Moreover, $Isol_{\vec{u},v,F}(\varphi)$ evaluates to $P(Isol_{\vec{u},v,F}(t_1), \ldots, Isol_{\vec{u},v,F}(t_k))$ and, hence, $P(d_1, \ldots, d_k)$ evaluates to false. The first part of the proof now yields that $t_i$ evaluates to $d_i$. Since $\varphi = P(t_1, \ldots, t_k)$, this means that $\varphi$ evaluates to false, as claimed.
- The case $\varphi = \neg P(t_1, \ldots, t_k)$ is analogous to the case $\varphi = P(t_1, \ldots, t_k)$ and therefore skipped.

*Induction step.* In the induction step, we distinguish between the two cases $\varphi = \varphi_1 \vee \varphi_2$ and $\varphi = \varphi_1 \wedge \varphi_2$.

- Let $\varphi = \varphi_1 \vee \varphi_2$ and assume that $Isol_{\vec{u},v,F}(\varphi)$ evaluates to false. Thus, both $Isol_{\vec{u},v,F}(\varphi_1)$ and $Isol_{\vec{u},v,F}(\varphi_2)$ evaluate to false. Applying the induction hypothesis yields that both $\varphi_1$ and $\varphi_2$ evaluate to false. Thus, $\varphi = \varphi_1 \vee \varphi_2$ evaluates to false, as claimed.
- The case $\varphi = \varphi_1 \wedge \varphi_2$ is analogous to the case $\varphi = \varphi_1 \vee \varphi_2$ and therefore skipped. □

We can also show (again using 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 $\vec{u}$.

Lemma 2.7. *Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a specification, $\vec{p} \in D^n$ an interpretation for $\vec{u}$, and $q \in D$ an interpretation for $v$ such that there is some interpretation for $\vec{x}$ that makes the formula $Isolate_{\vec{u},v,F}(\psi(f, \vec{x}))$ evaluate to false. Then, there exists no function $h$ satisfying the specification that maps $\vec{p}$ to $q$.*

Proof. Let $h$ be a function that satisfies the specification and maps $\vec{p}$ to $q$. Then, $\psi(f, \vec{x})$ evaluates to true for every interpretation of $\vec{x}$. By Lemma 2.6, this means that $Isolate_{\vec{u},v,F}(\psi(f, \vec{x}))$ always evaluates to true or $\bot$ (it cannot evaluate to false, because then $\varphi$ would evaluate to false as well). However, this is a contradiction to the assumption that there exists an interpretation for $\vec{x}$ on which the formula $Isolate_{\vec{u},v,F}(\psi(f,\vec{x}))$ evaluates to false. □

We can now define single-point refutable specifications.

*Definition 2.8 (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 $H : D^n \to D$ 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 $Isolate_{\vec{u},v,F}(\psi(f,\vec{x}))$ evaluates to false.

Intuitively, the above says that a specification is single-point refutable if whenever a hypothesis function $H$ does not satisfy 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, $\psi$ evaluates to *false* for some interpretation of $\vec{x}$ only assuming that $f(\vec{p}) = H(\vec{p})$.

In fact, single-point refutable specifications are single-point definable, which we formalize below.

Lemma 2.9. *If a specification $\forall\vec{x}.\ \psi(f,\vec{x})$ is single-point refutable, then it is single-point definable.*

Proof. Let $\forall\vec{x}.\ \psi(f,\vec{x})$ be a single-point refutable specification, and assume that it is not single-point definable. Moreover, let $\mathcal{F}$ be the class of all functions that satisfy this specification. Then, there exists a function $h' : D^n \to D$ such that for every input $\vec{p} \in D^n$, 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 2.7, there is no function that satisfies the specification and that maps $\vec{u}$ to $v$, which contradicts the fact that $h$ satisfies the specification. □

Let us illustrate the definition of single-point refutable specifications through an example and a non-example.

*Example 2.10.* Consider the following specifications in the first-order theory of arithmetic:

- The specification
$$\forall x,y.\ f(15,23) = 19 \wedge f(90,20) = 91 \wedge \ldots \wedge f(28,24) = 35$$
is single-point refutable. More generally, any set of input-output samples can be written as a conjunction of constraints that forms a single-point refutable specification.
- The specification
$$\forall x.\ f(0) = 0 \wedge f(x+1) = f(x) + 1$$
is *not* a single-point refutable specification, though it is single-point definable. Given a hypothesis function (e.g., $H(i) = 0$ for all $i$), the formula $f(x+1) = f(x) + 1$ 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 $\bot$ and, hence, the whole formula will never evaluate to false.)

When a specification $\forall\vec{x}.\ \psi(f,\vec{x})$ is single-point refutable, given an *expression* $H$ for $f$ that does not satisfy the specification, we can check satisfiability of the formula
$$\exists\vec{u}\,\exists v\exists\vec{x}.\ \big(v = H(\vec{u}) \wedge \neg Isolate_{\vec{u},v,F}(\psi(H/f,\vec{x}))\big).$$

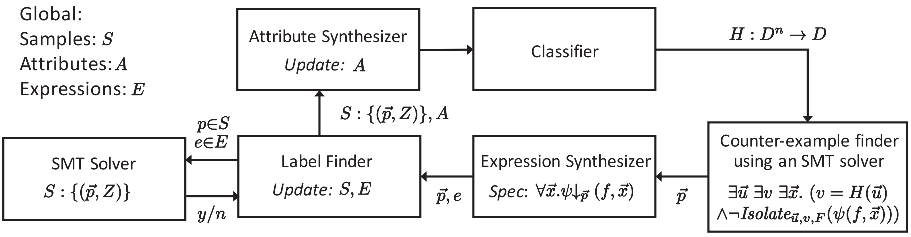
Global:
Samples: $S$
Attributes: $A$
Expressions: $E$
Attribute Synthesizer
Update: $A$
Classifier
$H : D^n \to D$
$S : \{(\vec{p}, Z)\}, A$
$p \in S$
$e \in E$
SMT Solver
$S : \{(\vec{p}, Z)\}$
Label Finder
Update: $S, E$
$y/n$
$\vec{p}, e$
Expression Synthesizer
Spec: $\forall \vec{x}. \psi{\downarrow_{\vec{p}}} (f, \vec{x})$
$\vec{p}$
Counter-example finder using an SMT solver
$\exists \vec{u}\ \exists v\ \exists \vec{x}.\ (v = H(\vec{u})$
$\wedge \neg Isolate_{\vec{u},v,F}(\psi(f, \vec{x})))$

Fig. 2. The general synthesis framework based on learning classifiers.

Assuming the underlying quantifier-free theory has a decidable satisfiability problem and one can construct models, the valuation of $\vec{u}$ gives a *concrete* input $\vec{p}$, and Lemma 2.7 shows that $H$ is definitely wrong on this input. This will form the basis of generating counterexample inputs in the synthesis framework that we present next.

## 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 2. 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 $Z \subseteq E$ of expressions that are correct for $\vec{p}$ (such a sample means that the specification allows mapping $\vec{p}$ to $e(\vec{p})$, for any $e \in Z$, but not to $e'(\vec{p})$, for any $e' \in E \setminus Z$).

***Phase 1.*** In every round, the classifier produces a hypothesis expression $H$ for $f$. The process starts with a simple expression $H$, such as the one that maps all inputs to a constant. We feed $H$ in every round to 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 2.8), 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 2.7). If $H$ satisfies the specification, then the formula would be unsatisfiable (by Lemma 2.6) 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

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

Intuitively, the above specification asks for a function $\widehat{f}$ that *"works"* for the input $\vec{p}$ (i.e., there exists a function $g$ satisfying the specification such that $g(\vec{p}) = \widehat{f}(\vec{p})$). We do this by first constructing the formula that isolates the specification to $\vec{u}$ with output $v$ and demand that the specification evaluates 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 maps $\vec{p}$ to a value that is consistent with the original specification, which we formalize next.

Lemma 3.1. *Let $\psi(f,\vec{x})$ be a single-point refutable specification and $\mathcal{F}$ the class of all functions satisfying $\psi(f,\vec{x})$. Moreover, let $\vec{p} \in D^n$ be an input to $f$, and let $e$ be a solution to the synthesis problem with specification $\psi{\downarrow_{\vec{p}}}\ (\widehat{f},\vec{x})$. Then, there exists a function $g \in \mathcal{F}$ such that $g(\vec{p}) = e(\vec{p})$.*

Proof. Using similar arguments as in the proof of Lemma 2.6, we can show that if $\mathit{Isol}_{\vec{u},v,T}(\varphi)$ (and, hence, $\mathit{Isolate}_{\vec{u},v,T}(\varphi)$) evaluates to true on some valuation of the free variables $\vec{x}$, $\vec{u}$, and $v$, then the formula $\varphi$ also evaluates to true on the valuation for $\vec{x}$. Thus, by substituting $\vec{p}$ for $\vec{u}$ in $\mathit{Isolate}_{\vec{u},v,T}(\psi(f,\vec{x}))$, we know that if $\mathit{Isolate}_{\vec{u},v,T}(\psi(f,\vec{x}))[\vec{p}/\vec{u}]$ evaluates to true, then $\psi$ evaluates to true if $f$ maps $\vec{p}$ to $v$. Moreover, by substituting $\widehat{f}(\vec{p})$ for $v$, we obtain the specification $\psi{\downarrow_{\vec{p}}}\ (\widehat{f},\vec{x})$, which constraints $\widehat{f}$ such that $\psi(f,\vec{x})$ to evaluates to true if $f(\vec{p}) = \widehat{f}(\vec{p})$. Thus, any solution $e$ to the synthesis problem with specification $\psi{\downarrow_{\vec{p}}}\ (\widehat{f},\vec{x})$ guarantees that $\psi(f,\vec{x})$ evaluates to true if $f(\vec{p}) = e(\vec{p})$.

Now, let $h \in \mathcal{F}$ some function satisfying $\psi$. Since $\psi$ is single-point refutable (and, hence, also single-point definable), the function

$$g(\vec{x}) = \begin{cases} e(\vec{p}) & \text{if } \vec{x} = \vec{p}\text{; and} \\ h(\vec{x}) & \text{otherwise} \end{cases}$$

also satisfies the specification. Thus, $g$ is a function satisfying $g \in \mathcal{F}$ and $g(\vec{p}) = e(\vec{p})$. □

We emphasize that this new synthesis problem is simpler than the original problem (since it only requires to synthesize an expression for a single input) and that we can use any (existing) expression synthesizer to solve it. One important challenge in this context clearly is to synthesize an expression that is "good" (or general) in the sense that it works for all (or at least many) inputs in the region $\vec{p}$ belongs to. One possible way to achieve this is to apply Occam's razor principle and synthesize an expression that is as simple as possible with respect to some total order of the expressions (e.g., the length of the expression or the maximal nesting of sub-expressions). Another way is to define a distance metric on inputs and synthesize an expression that does not only work for the input $\vec{p}$ but also for other known inputs in the sample $S$ whose distance to $\vec{p}$ is small. We describe these two approaches and various further heuristics in more detail in Section 5.1, where we present our implementation of a synthesizer for linear integer arithmetic expressions.

***Phase 3.*** Once we synthesize an expression $e$ that works for $\vec{p}$, we feed it to the next phase, which adds $e$ to the set of all expressions $E$ (if $e$ is new) and adds $\vec{p}$ to the set of samples. It then proceeds to find the set of *all* expressions in $E$ that work for all the inputs in the samples, and computes the new set of samples. 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). To do this, we need to split the input domain into *regions* defined by a set 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 set of predicates. Two samples $(\vec{j}, E_1)$ and $(\vec{j}', E_2)$ are said to be *inseparable* if for every predicate $p \in A$, $p(\vec{j}) \equiv p(\vec{j}')$. 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, if every subset $T \subseteq S$, say $T = \{(\vec{i_1}, E_1), (\vec{i_2}, E_2), \ldots (\vec{i_t}, E_t)\}$, where every pair of inputs in $T$ is inseparable, then $\bigcap_{i=1}^{t} E_i \neq \emptyset$. 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 that 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 predicates to define the regions and expressions at leaves 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. (We refer to standard textbooks on machine learning, e.g., Mitchell (1997), for more information on decision tree learning.) For the purpose of learning the classifier, we assume samples to be vectors of the Boolean values $\mathbb{B} = \{F, T\}$ (these encode the values of the various attributes on the counterexample input returned). The more general case that datapoints also contain rational numbers can be handled in a straightforward manner as in Quinlan's C 5.0 algorithm (Quinlan 1993; RuleQuest Research 2015).

To make the learning problem precise, let us fix a finite set $L = \{\lambda_1, \ldots, \lambda_k\}$ of labels with $k \geq 2$, and let $\vec{x}_1, \ldots, \vec{x}_m \in \mathbb{B}^n$ be $m$ individual inputs (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 (2002)), is defined as follows.

*Definition 4.1 (Disjoint Multi-Label Learning Problem).* Given a finite training set $S = \{(\vec{x}_1, Y_1), \ldots, (\vec{x}_m, Y_m)\}$ where $Y_i \subseteq L$ and $Y_i \neq \emptyset$ for every $i \in \{1, \ldots, m\}$, the *disjoint multi-label learning problem* is to find a decision tree classifier $h : \mathbb{B}^n \to L$ 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 in the training set), in contrast to classical machine-learning settings where classifier are allowed to make (small) errors.

**ALGORITHM 1:** Multi-label Decision Tree Learning Algorithm

```
   Input: A finite set $S \subseteq \mathbb{B}^n \times 2^L$ of datapoints.
1  return DecTree ($S$, $\{1, \dots, n\}$).
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 (i.e., there exists a label $\lambda$ such that $\lambda \in Y$ for each
       $(\vec{x}, Y) \in S$) then
5          Select a common label $\lambda$ and return the single-node tree $r$ with label $\lambda$.
6      else
7          Select an attribute $i \in A$ that (heuristically) best splits the sample $S$.
8          Split $S$ into $S_i = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = F\}$.
9          Label $r$ with attribute $i$ and return the tree with root node $r$, left subtree DecTree ($S_i$, $A \setminus \{i\}$),
           and right subtree DecTree ($S_{\neg i}$, $A \setminus \{i\}$).
10     end
```

We use a straightforward modification of Quinlan's C 5.0 algorithm (Quinlan 1993; RuleQuest Research 2015) to solve the disjoint multi-label learning problem. This modification, sketched in pseudo code 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 $i \in \{1, \dots, n\}$ and splits the set into two disjoint, nonempty subsets $S_i = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = F\}$ (we explain shortly how the attribute $i$ 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 at least one 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 $\lambda$ (arbitrarily), constructs a single-node tree that is labeled with $\lambda$, and returns from the recursion. However, it might happen during construction that a set of datapoints does not 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 by synthesizing new attributes whenever necessary).

The following theorem states the correctness of Algorithm 1 (i.e., that the algorithm indeed produces a solution to the disjoint multi-label learning problem), which is a result independent of the exact way an attribute is chosen.

Theorem 4.2. *Let $L = \{\lambda_1, \dots, \lambda_k\}$ be a set of labels with $k \geq 2$ and $S = \{(\vec{x}_1, Y_1), \dots, (\vec{x}_m, Y_m)\} \subseteq \mathbb{B}^n \times 2^L$ a finite training set where $Y_i \neq \emptyset$ for every $i \in \{1, \dots, n\}$. Moreover, assume that each two distinct datapoints $(\vec{x}, Y), (\vec{x}', Y') \in S$ can be separated by some attribute $i \in \{1, \dots, n\}$ (i.e., $\vec{x}[i] \neq \vec{x}'[i]$). Then, Algorithm 1 terminates and returns a decision tree classifier $h : \mathbb{B}^n \to L$ that satisfies $h(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S$.*

Proof. We show Theorem 4.2 by induction over the construction of the tree.

*Base case.* Assume that the function DecTreeis called with a set $S$ of datapoint that share a common label (i.e., that there exists a label $\lambda$ such that $\lambda \in Y$ for each $(\vec{x}, Y) \in S$). Then, the condition in Line 4 evaluates to true and the algorithm returns a decision tree $h$ consisting of a single node that is labeled with a label shared by all datapoints of $S$ (see Line 5). Thus, $h$ satisfies $h(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S$.

*Induction step.* Assume that the function DecTreeis called with a set $S$ of datapoints that do not share a common label and a set $A$ of available attributes. Then, the condition in Line 4 is false, and the algorithm proceeds with Lines 7 to 9.

First, we observe that $A \neq \emptyset$: since we assume that all datapoints can be separated by an attribute, $A = \emptyset$ implies that $S$ is a singleton and, hence, the condition in Line 4 would be true. Thus, the algorithm can pick an attribute $i \in A$ (Line 7), partition $S$ into two subsamples $S_i, S_{\neg i}$ (Line 8), and recursively constructs the decision trees $h_i = \texttt{DecTree}(S_i, A \setminus \{i\})$ and $h_{\neg i} = \texttt{DecTree}(S_{\neg i}, A \setminus \{i\})$ (Line 9). Finally, it returns the decision tree $h$ with root node $r$ whose subtrees are $h_i$ and $h_{\neg i}$, respectively.

Since the root of $h$ is labeled with attribute $i$, one can write $h$ as

$$h(\vec{x}) = \begin{cases} h_i(\vec{x}) & \text{if } \vec{x}[i] = T\text{; and} \\ h_{\neg i}(\vec{x}) & \text{if } \vec{x}[i] = F. \end{cases}$$

Moreover, applying the induction hypothesis yields that $h_i$ satisfies $h_i(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S_i$ and $h_{\neg i}(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S_{\neg i}$. Thus, if $\vec{x}[i] = T$ for some $(\vec{x}, Y) \in S$, then $(\vec{x}, Y) \in S_i$ and, hence, $h(\vec{x}) = h_i(\vec{x}) \in Y$; on the other hand, if $\vec{x}[i] = F$ for some $(\vec{x}, Y) \in S_{\neg i}$, then $(\vec{x}, Y) \in S_{\neg i}$ and, hence, $h(\vec{x}) = h_{\neg i}(\vec{x}) \in Y$. □

The selection of a “good” attribute to split 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. This problem is best understood in the simpler single-label setting in which datapoints are labeled with one out of two possible labels, say 0 or 1. To obtain a small decision tree, the learning algorithm should split samples such that the resulting subsamples are as pure as possible (i.e., one subsample contains as many datapoints as possible labeled with 0, whereas the other subsample contains as many datapoints as possible labeled with 1). This way, the learner will quickly arrive at samples that contain a single label and, hence, produce a small tree.

The quality of a split can be formalized by the notion of a *measure*, which, roughly, is a measure $\mu$ mapping pairs of 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 (Hastie et al. 2001). In the case of multi-label classifiers, however, finding a good measure is still a matter of ongoing research (e.g., see Tsoumakas and Katakis (2007) for an overview). 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. More precisely, the main idea is to replace each multi-labeled datapoint $(\vec{x}, \{\lambda_1, \ldots, \lambda_k\})$ with the datapoints $(\vec{x}, \lambda_1), \ldots, (\vec{x}, \lambda_k)$ and proceeds as in classical decision tree learning.

We now briefly sketch these modifications, including a modification described by Clare and King (2001). In all cases, we fix $R = \mathbb{R}$ and let $\preceq$ be 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 $p_\lambda$ is the relative frequency of the label $\lambda$ defined by

$$p_\lambda = \frac{|\{(\vec{x}, Y) \in S \mid \lambda \in Y\}|}{\sum_{(\vec{x}, Y) \in S} |Y|}.$$

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 $p_\lambda$ is again the relative frequency of the label $\lambda$ (see above). The Gini measure $\mu_g(S_1, S_2)$ is the weighted average of $g(S_1)$ and $g(S_2)$.

*pq-entropy.* The modification of entropy by Clare and King (2001) accounts for multiple labels by considering for each label the probability of being labeled with $\lambda$ (i.e., the relative frequency of $\lambda$) as well as probability of not being labeled with $\lambda$. 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 $p_\lambda$ is the relative frequency of the label $\lambda$ and $q_\lambda = 1 - p_\lambda$. As measure $\mu_{pq\text{-}e}(S_1, S_2)$, Clare and King use the weighted average of $pq\text{-}e(S_1)$ and $pq\text{-}e(S_2)$.

However, all of these approaches share the disadvantage that the association of datapoints to sets of labels is lost. As a consequence, measures can be high even if all datapoints share a common label; for instance, such a situation occurs for $S = \{(\vec{x}_1, Y_1), \ldots, (\vec{x}_m, Y_m)\}$ with $\{\lambda_1, \ldots, \lambda_\ell\} \subseteq Y_i$ for every $i \in \{1, \ldots, m\}$. Therefore, one would ideally like to have a measure 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. We now present a measure, based on the combinatorial problem of finding minimal hitting sets, that possesses this property. To the best of our knowledge, this measure is a novel contribution and has not been studied in the literature.

For a set $S$ of datapoints, a set $H \subseteq L$ is a *hitting set* if $H \cap Y \neq \emptyset$ for each $(\vec{x}, Y) \in S$. Moreover, we define the measure $hs(S) = \min_{\text{hitting set } H} |H| - 1$ (i.e., 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

$$\mu_{hs}(S_1, S_2) = \Big(\max\{hs(S_1), hs(S_2)\}, hs(S_1) + hs(S_2)\Big).$$

Unfortunately, computing $hs(S)$ is computationally hard. Therefore, we implemented a standard greedy algorithm (the dual of the standard greedy set cover algorithm (Chvatal 1979)), which runs in time polynomial in the size of the sample and whose solution is at most logarithmically larger than the optimal solution.

## 5 A SYNTHESIS ENGINE FOR LINEAR INTEGER ARITHMETIC

We now describe an instantiation of our framework (described in Section 3) for synthesizing functions expressible in linear integer arithmetic against quantified linear integer arithmetic specifications.

The counterexample input finder (Phase 1) and the computing of labels for counterexample inputs (Phase 3) are implemented straightforwardly using an SMT solver (note that the respective formulas will be in quantifier-free linear integer arithmetic). 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 $\perp$ 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 $D$ (which in our case integers) itself.

### 5.1 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.

In the first phase, it checks whether one of the existing expressions in the global set $E$ already works for $\vec{p}$. This is done by calling the label finder (as in Phase 3). If none of the expressions in $E$ work for $\vec{p}$, then the expression synthesizer proceeds to the second phase, where it generates a new synthesis problem with specification $\forall \vec{x}.\ \psi{\downarrow_{\vec{p}}}(\hat{f}, \vec{x})$ 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 $\forall \vec{x}.\ \psi(f, \vec{x})$ be a specification with a function symbol $f : \mathbb{Z}^n \to Z$, which is to be synthesized, and universally quantified variables $\vec{x} = (x_1, \dots, x_m)$. Our algorithm synthesizes affine expressions of the form $(\sum_{i=1}^{n} a_i \cdot y_i) + b$ where $y_1, \dots, y_n$ are integer variables, $a_i \in \mathbb{Z}$ for $i \in \{1, \dots, n\}$, 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 $\psi$ 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, then the algorithm has identified a correct expression, returns it, and terminates. If this not the case, then 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. The synthesizer and verifier are implemented as follows.

*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 $\psi$ 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 $\vec{a} = (a_1, \dots, a_n)$, 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

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

Note that $\varphi$ 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 $\varphi$; note that $\varphi$ 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

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

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, then an expression satisfying the specification has been found and the synthesizing algorithm returns it.

*5.1.1 Further Heuristics.* Our implementation for synthesizing expressions for linear arithmetic constraints has several other heuristics that we briefly describe.

First, some technical aspects of the counterexample finder and the label finder are implemented a bit differently than explained above. We use array theories and uninterpreted functions to extract the counterexample point from the specification and the hypothesis (instead of using the Isolate transformer), and to check if a point works for an expression.

The counterexample finder prioritizes data-points that have a single classification, and returns multiple counterexamples in each round, to facilitate better learning.

We also 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. If none of these expression work for the counterexample point, then the expression synthesizer is invoked.

There are also phases in our algorithm where, when examining an enumerated expression, we would ask a constraint solver whether this expression would work for any input (not necessarily the counterexample input) and if possible, add the new point and the expression to the sample. Further, when we do find a unique expression that works for the counterexample, we ask the constraint solver for *more* points for which this expression would work, and add them as extra samples. When multiple expressions work for a point, we strive to find another point for which only one of them works (to avoid considering spurious expressions) and add them to the sample.

The expression synthesizer also uses a heuristic motivated by a geometric intuition. We would expect the correct expression for a point to work also on points neighboring it (unless it lies close to the boundary of a piece-wise region). To synthesize a $d$-dimension expression, we need at least $(d + 1)$ points that lie on that plane. If $(y_1, y_2, \ldots y_d)$ is a point, then it is highly likely that the "correct" expression for the point would also work for 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 constrain the SMT solver to synthesize a $d$-dimensional expression with integer coefficients that works for all these $(d + 1)$ points. If no such expression exists, then we resort to synthesizing an expression only for the counterexample.

## 5.2 Predicate Synthesizer

Since the decision tree learning algorithm (which is our classifier) copes extremely well with a large number of attributes, we do not spend time in generating a small set of predicates. We build an *enumerative* predicate synthesizer that 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 $\Sigma_{i=1}^{n} |a_i| \leq q$, and $|b| \leq q^2$. If $A_q$ is already adequate for $S$ (which can be checked by recursively splitting the sample with respect to each predicate in $A_q$ and checking if all samples at each leaf has a common label), then we stop, or we increase the parameter $q$ by one and iterate. 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 (assuming each individual variable is also an attribute) provided $q$ is large enough.

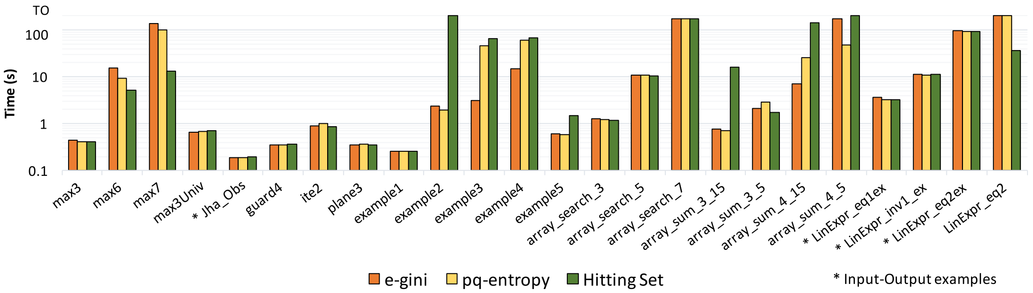

Fig. 3. Experimental results.

### 5.3 Classifier 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 $\tau$, 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 $\tau$ 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 $\textit{to-ite}(v) := e$; and
- if $v$ is an inner node labeled with predicate $p$ and children $v_1$ and $v_2$, then $\textit{to-ite}(v) := \textit{ite}(p, \textit{to-ite}(v_1), \textit{to-ite}(v_2))$.

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

## 6 EVALUATION

We implemented the framework described in Section 5 for specifications written in the SyGuS format (Raghothaman and Udupa 2014; Alur et al. 2015). The implementation is about 5K lines in C++ with API calls to the Z3 SMT solver (De Moura and Bjørner 2008).

We evaluated our tool parameterized using the different measures in Section 4 against 44 benchmarks. These benchmarks are predominantly from the 2014–2016 SyGuS competitions (Alur et al. 2015, 2016a, 2016b). Additionally, there is an example from Jha et al. (2010) 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, `Jha_Obs`, from it). The synthesis specification `max3Univ` reformulates the specification for `max3` using universal quantification, as

$$\forall x, r_1, r_2, y_1, y_2, y_3.\ (r_1 < 0 \land r_2 < 0) \Rightarrow$$
$$((y_1{=}x \land y_2{=}x{+}r_1 \land y_3{=}x{+}r_2) \lor (y_1{=}x{+}r_1 \land y_2{=}x \land y_3{=}x{+}r_2) \lor (y_1{=}x{+}r_1 \land y_2{=}x{+}r_2 \land y_3{=}x))$$
$$\Rightarrow \textit{max}3(y_1, y_2, y_3) = x.$$

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 200s timeout.

Table 1 and Figure 3 compare the three measures: *e-gini*, *pq-entropy*, and *hitting set*. None of the algorithms dominates. All solvers time-out on two benchmarks each. 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.

Table 1 also compares the compositional approach to synthesis presented in this article (using three learners, one for leaf expressions, one for predicates, and one for the Boolean expression combining them) with a *monolithic* learner based on a CEGIS algorithm that synthesizes the entire function using a constraint solver. As is evident from this table, our algorithm is more often than not faster than the monolithic constraint-based solver. The latter times out on a large number of specifications.

The CVC4 SMT-solver based synthesis tool (Reynolds et al. 2015) (which won the conditional linear integer arithmetic track in the SyGuS 2015 and 2016 competitions (Alur et al. 2016a, 2016b)) worked very fast on these benchmarks, in general, but does not *generalize* from underspecifications. On specifications that list a set of input-output examples (marked with $*$ in Figure 3), 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) renders them unsolvable. CVC4 was also not able to solve, surprisingly, the fairly simple specification `max3Univ` (although it has the single-invocation property (Reynolds et al. 2015)).

The general track SyGuS solvers (enumerative, stochastic, constraint-solver, and Sketch) (Alur et al. 2015) do not work well for these benchmarks (and did not fare well in the competitions either); for example, the enumerative solver, which was the winner in 2014 can solve only 16 of the 44 benchmarks.

The above results show that the synthesis framework developed in this article that uses theory-specific solvers for basic expressions and predicates, and combines them using a classification learner yields a competitive solver for the linear integer arithmetic domain. We believe more extensive benchmarks are needed to fine-tune our algorithm, and especially in choosing the right statistical measures for decision-tree learning.

## 7 CONCLUSIONS

We have presented a novel compositional framework for synthesizing piece-wise functions over any theory that combines three engines—a synthesizer for the simpler leaf expressions used in a region, the predicates that can be used to define the boundaries of regions, and the Boolean expression that defines the regions themselves and chooses the leaf expressions to apply to each region. We have shown how to formulate automatically the specifications for synthesizing leaf expressions and predicate expressions from the synthesis specification, and developed generic classification algorithms for learning regions and mapping them to expressions using decision-tree based machine-learning algorithms.

One future direction worth pursuing is to build both specific learning algorithms for synthesis problems based on our framework, as well as build general solutions to synthesis (say for all SyGuS specifications). One piece of work that has emerged since the publication of our result is the EUSolver (Alur et al. 2017), which can be seen as an instantiation of our framework, using enumerative techniques to synthesize both leaf expressions and predicates, and using a decision-tree classifier similar to ours for finding and mapping regions to expressions. The EUSolver has performed particularly well in the SyGuS 2016 (Alur et al. 2016b) competition, winning several tracks, and in particular working well for the class of ICFP benchmark synthesis challenge problems, solving a large proportion of them for the first time. We also note that the original winner for the ICFP benchmarks (in the competition held in 2013 (Akiba et al. 2013)) also used

Table 1. Experimental Performance of the Measures e-gini, pq-entropy, Hitting Set, and the Constraint Solver

| Benchmarks | e-gini | | pq-entropy | | Hitting set | | Constr. solver |
|---|---|---|---|---|---|---|---|
| | Rounds | Time | Rounds | Time | Rounds | Time | Time |
| Jha_Obs | 1 | 0.2 | 1 | 0.2 | 1 | 0.2 | 0.5 |
| LinExpr_eq1 | — | TO | — | TO | — | TO | TO |
| LinExpr_eq1ex | 9 | 3.6 | 10 | 3.3 | 10 | 3.2 | 122.9 |
| LinExpr_eq2 | — | TO | — | TO | 31 | 36.8 | 2.3 |
| LinExpr_eq2ex | 48 | 94.5 | 49 | 93.6 | 50 | 93.1 | 1.0 |
| LinExpr_inv1_ex | 39 | 11.5 | 39 | 10.9 | 38 | 11.1 | 0.1 |
| array_search_2 | 7 | 0.6 | 7 | 0.6 | 6 | 0.6 | 2.6 |
| array_search_3 | 7 | 1.3 | 7 | 1.2 | 7 | 1.2 | 30.2 |
| array_search_4 | 12 | 4.5 | 11 | 4.4 | 16 | 4.6 | TO |
| array_search_5 | 19 | 10.7 | 20 | 10.7 | 16 | 10.6 | TO |
| array_search_6 | 22 | 50.2 | 22 | 40.0 | 22 | 49.3 | TO |
| array_search_7 | 27 | 174.6 | 22 | 174.0 | 20 | 170.1 | TO |
| array_sum_2_15 | 5 | 0.3 | 5 | 0.3 | 5 | 0.3 | 0.7 |
| array_sum_2_5 | 7 | 0.4 | 7 | 0.4 | 12 | 0.6 | 0.2 |
| array_sum_3_15 | 9 | 0.8 | 9 | 0.7 | 40 | 15.8 | 43.0 |
| array_sum_3_5 | 31 | 2.1 | 39 | 2.9 | 20 | 1.7 | 12.7 |
| array_sum_4_15 | 28 | 7.0 | 76 | 26.0 | 44 | 139.5 | TO |
| array_sum_4_5 | 187 | 169.8 | 105 | 47.2 | — | TO | TO |
| max2 | 3 | 0.2 | 3 | 0.2 | 3 | 0.2 | 0.3 |
| max3 | 8 | 0.4 | 8 | 0.4 | 8 | 0.4 | 4.7 |
| max3Univ | 9 | 0.7 | 9 | 0.7 | 8 | 0.7 | 2.5 |
| max4 | 18 | 1.0 | 16 | 0.9 | 12 | 0.7 | TO |
| max5 | 44 | 2.8 | 51 | 3.4 | 32 | 2.2 | TO |
| max6 | 130 | 15.4 | 94 | 9.3 | 47 | 5.2 | TO |
| max7 | 327 | 136.1 | 271 | 98.6 | 65 | 13.2 | TO |
| max8 | — | TO | — | TO | 140 | 92.0 | TO |
| example1 | 3 | 0.3 | 3 | 0.3 | 3 | 0.3 | 1.2 |
| example2 | 31 | 2.4 | 30 | 2.0 | — | TO | TO |
| example3 | 10 | 3.2 | 12 | 46.8 | 14 | 65.3 | 2.6 |
| example4 | 29 | 15.0 | 46 | 61.1 | 52 | 66.7 | TO |
| example5 | 9 | 0.6 | 9 | 0.6 | 20 | 1.5 | TO |
| guard1 | 3 | 0.3 | 3 | 0.3 | 3 | 0.3 | 0.2 |
| guard2 | 2 | 0.3 | 2 | 0.3 | 2 | 0.3 | 0.2 |
| guard3 | 4 | 0.4 | 4 | 0.4 | 4 | 0.4 | 0.3 |
| guard4 | 4 | 0.4 | 4 | 0.4 | 4 | 0.4 | 0.3 |
| ite1 | 4 | 0.7 | 4 | 0.7 | 4 | 0.7 | 2.8 |
| ite2 | 9 | 0.9 | 11 | 1.0 | 7 | 0.9 | 1.9 |
| plane1 | 1 | 0.2 | 1 | 0.2 | 1 | 0.2 | 0.1 |
| plane2 | 1 | 0.4 | 1 | 0.4 | 1 | 0.4 | 0.2 |
| plane3 | 1 | 0.4 | 1 | 0.4 | 1 | 0.4 | 0.2 |
| s1 | 5 | 0.3 | 5 | 0.3 | 5 | 0.3 | 0.1 |
| s2 | 2 | 0.2 | 2 | 0.2 | 2 | 0.2 | 0.1 |
| s3 | 1 | 0.2 | 1 | 0.2 | 1 | 0.2 | 0.1 |

Times are given in seconds. “TO” indicates a timeout of 200s.

a compositional approach to synthesis that discovered leaf expressions individually for points and then combined them. This experimental evidence suggests that the compositional framework outlined in this article is likely a more efficient approach to synthesis of piece-wise functions.

Another interesting future direction is to extend our framework beyond single-point definable/refutable specifications, in particular, to bounded point definable/refutable functions. When synthesizing *inductive invariants* for programs, the counterexamples to hypothesized invariants are not single counterexamples, but usually involve *two* counterexamples connected with an implication (see the model of ICE learning (Garg et al. 2014b)). Extending our framework to synthesize for such specifications would be interesting. (Note that in invariants, the leaf expressions are fixed (T/F), and only the predicates separating regions need to be synthesized.)

In summary, the synthesis approach developed in this article brings a new technique, namely machine learning, to solving synthesis problems, in addition to existing techniques such as enumeration, constraint-solving, and stochastic search. Leaf expressions and predicates belong to particular theories that have complex semantics, and are hence best synthesized using dedicated synthesis procedures. However, combining the predicates to form regions and mapping regions to particular expressions can be seen as a generic classification problem that can be realized by learning Boolean formulas, and is independent of the underlying theories. By using a learner of Boolean formulas (decision trees in our setting), we can combine theory-specific synthesizers for leaf expressi ons and predicates to build efficient learners of piece-wise functions.

## ACKNOWLEDGMENTS

A shorter version of this article was presented at the conference TACAS (Neider et al. 2016). This material is based upon work supported by the National Science Foundation under Grants No. 1138994 and No. 1527395.

## REFERENCES

Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michał Moskal, and Nikhil Swamy. 2013. Calibrating research in program synthesis using 72,000 hours of programmer time. *MSR, Redmond, WA, Tech. Rep* (2013).

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25.

Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated grading of DFA constructions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'13)*. IJCAI/AAAI.

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016a. Results and analysis of SyGuS-Comp15. *Electro. Proc. Theoret. Comput. Sci.* 202 (2016), 326.

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016b. SyGuS-Comp 2016: Results and analysis. *Electro. Proc. Theoret. Comput. Sci.* 229 (2016), 178202.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 319–336.

Rajeev Alur and Nimit Singhania. 2014. Precise piecewise affine models from input-output data. In *Proceedings of the International Conference on Embedded Software (EMSOFT'14)*. ACM, 3:1–3:10.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5.* Technical Report. Department of Computer Science, The University of Iowa. Retrieved from http://www.SMT-LIB.org.

Alberto Bemporad, Andrea Garulli, Simone Paoletti, and Antonio Vicino. 2005. A bounded-error approach to piecewise affine system identification. *IEEE Trans. Automat. Contr.* 50, 10 (2005), 1567–1580.

Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C. Myers. 2014. Using program analysis to improve database applications. *IEEE Data Eng. Bull.* 37, 1 (2014), 48–59.

V. Chvatal. 1979. A greedy heuristic for the set-covering problem. *Math. Operat. Res.* 4, 3 (1979), 233–235.

Amanda Clare and Ross D. King. 2001. Knowledge discovery in multi-label phenotype data. In *Proceedings of the Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'01) (LNCS)*, Vol. 2168. Springer, 42–53.

Leonardo De Moura and Nikolaj Bjørner. 2008. 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 (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, 337–340.

Giancarlo Ferrari-Trecate, Marco Muselli, Diego Liberati, and Manfred Morari. 2003. A clustering technique for the identification of piecewise affine systems. *Automatica* 39, 2 (2003), 205–217.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014a. ICE: A robust framework for learning invariants. In *Proceedings of the Conference on Computer Aided Verification (CAV'14) (LNCS)*, Vol. 8559. Springer, 69–87.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014b. ICE: A robust framework for learning invariants. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 69–87.

Pranav Garg, P. Madhusudan, Daniel Neider, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'16)*. ACM, 499–512.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'11)*. ACM, 317–330.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer, New York.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1 (ICSE'10)*. ACM, New York, NY, 215–224.

Rong Jin and Zoubin Ghahramani. 2002. Learning with multiple labels. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS'02)*. MIT Press, 897–904.

Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of Onward!, part of SLASH'14*. ACM, 173–184.

Emanuel Kitzelmann. 2010. Inductive programming: A survey of program synthesis techniques. In *Proceedings of the American Institute of Physics (AAIP'09), Revised Papers (LNCS)*, Vol. 5812. Springer, 50–73.

Christof Löding, P. Madhusudan, and Daniel Neider. 2016. Abstract learning frameworks for synthesis. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16) (LNCS)*, Vol. 9636. Springer, to appear.

Jedidiah McClurg, Hossein Hojjat, Pavol Cerný, and Nate Foster. 2015. Efficient synthesis of network updates. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 196–207.

Tom M. Mitchell. 1997. *Machine Learning*. McGraw-Hill.

Daniel Neider, Shambwaditya Saha, and P. Madhusudan. 2016. Synthesizing piece-wise functions by learning classifiers. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 186–203.

Simone Paoletti, Aleksandar Lj. Juloski, Giancarlo Ferrari-Trecate, and René Vidal. 2007. Identification of hybrid systems: A tutorial. *Eur. J. Control* 13, 2-3 (2007), 242–260.

J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Mukund Raghothaman and Abhishek Udupa. 2014. Language to specify syntax-guided synthesis problems. *arXiv:1405.5590* (2014).

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proceedings of the Conference on Computer Aided Verification (CAV'15) (LNCS)*, Vol. 9207. Springer, 198–216.

RuleQuest Research. 2015. Data Mining Tools See5 and C 5.0. Retrieved from https://www.rulequest.com/see5-info.html.

Shambwaditya Saha, Pranav Garg, and P. Madhusudan. 2015a. Alchemist: Learning guarded affine functions. In *Proceedings of the Conference on Computer Aided Verification (CAV'15) (LNCS)*, Vol. 9206. Springer, 440–446.

Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. 2015b. NetGen: Synthesizing data-plane configurations for network policies. In *Proceedings of the Symposium on SDN Research (SOSR'15)*. ACM, 17:1–17:6.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 15–26.

Armando Solar-Lezama. 2013. Program sketching. *Software Tools Technol. Transfer* 15, 5–6 (2013), 475–495.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. ACM, 404–415.

Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *Int. J. Data Warehous. Min.* 3, 3 (2007), 1–13.

R. Vidal, S. Soatto, Yi Ma, and S. Sastry. 2003. An algebraic geometric approach to the identification of a class of linear hybrid systems. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, Vol. 1. 167–172.

Received November 2016; revised September 2017; accepted November 2017