Extending enumerative function synthesis via SMT-driven classification

Haniel Barbosa, Andrew Reynolds, Daniel Larraz, Cesare Tinelli
The University of Iowa, Iowa City, USA

Abstract—**Many relevant problems in formal methods can be tackled using enumerative syntax-guided synthesis (SyGuS). Algorithms for enumerative SyGuS range from universally applicable techniques based on counterexample-guided inductive synthesis (CEGIS), to more scalable but specialized techniques based on divide and conquer. This paper presents a novel algorithm for enumerative SyGuS, Unif+PI, which reaps the benefits of scalability based on divide and conquer without sacrificing generality. In this algorithm, an instance of an SMT solver is used as both a classifier and an attribute generator. Logical constraints in the form of test cases for the function-to-synthesize and failed classification attempts guide its search for new candidate solutions. We implement our approach as an extension of the CVC4SY solver and evaluate it on standard SyGuS benchmarks from different applications. We show that the new algorithm leads to significant gains in invariant synthesis with respect to state-of-the-art SyGuS solvers, and is competitive with state-of-the-art *k*-induction based model checking.**

I. Introduction

Syntax-guided synthesis (SyGuS) [1] is a recent paradigm for program synthesis, successfully used for a number of applications in formal verification and programming languages. It is characterized by supplementing the original synthesis problem with syntactic restrictions on the solution space, thus reducing the search effort. Most SyGuS solvers employ the *counterexample-guided inductive synthesis* (CEGIS) approach [31, 32]: a refinement loop in which a learner proposes solutions, and a verifier, usually an SMT solver [5], checks them, providing counterexamples for failures. Generally, the learner enumerates some set of terms, pruning those that it recognizes as spurious [33]. Despite its scalability issues, the simplicity and efficacy of enumerative SyGuS have made it the de facto approach for SyGuS, although alternatives exist for restricted fragments [2, 4, 26].

In the context of *point-wise* specifications, in which an input point is only related to its output and not to outputs of other input points, one such alternative is Alur et al.'s *divide and conquer* (D&C) enumeration [4]. Their key idea is to independently enumerate smaller terms that are correct on subsets of inputs and predicates that distinguish these subsets. Unifying terms via predicates into a conditional term correct on all inputs is seen as a classification problem, solved via decision tree learning. Their approach allows large solutions to be built from smaller terms, which can lead to significant performance gains by reducing the enumeration effort. However, while some important applications fall within this fragment, such as programming-by-examples (PBE) problems [15, 16], many interesting fragments, such as invariant synthesis, do not, as well as any function synthesis problem in which determining the value of the function on a given input cannot be made independently of its values on other inputs.

We present Unif+PI (Section III), a new algorithm that allows D&C to be applied on arbitrary specifications, independently of them being point-wise. To cope with the resulting more complex classification problem we encode it in SMT and use an SMT solver to both enumerate suitable terms and predicates as well as to build classifiers. We also present a variation (Section IV) of Unif+PI, which we denominate Unif+PI+E, that enumerates predicates independently, using the SMT solver only to enumerate suitable terms and relying on heuristic decision tree learning to build the classifier. We have implemented both approaches in the state-of-the-art SyGuS solver CVC4SY [25]. Our experimental evaluation (Section V) shows improvements over the state-of-the-art in SyGuS invariant synthesis and competitiveness against state-of-the-art *k*-induction techniques from model checking.

II. Conventions and background

We work in the context of many-sorted first-order logic with equality modulo theories (see, e.g., [9]) and assume the reader is familiar with the notions of signature, terms, and so on. In particular, we use the predicate symbol $\simeq$ for *term equality*, while the symbol $=$ stands for syntactic equality. The set of all terms occurring in a formula φ (resp. term t) is denoted by $\mathbf{T}(\varphi)$ (resp. $\mathbf{T}(t)$). A *theory* is a pair $T = (\Sigma, I)$ where Σ is a signature and I is a non-empty class of Σ-interpretations, the *models of* T, that is closed under variable reassignment and isomorphism. A Σ-formula φ is T-satisfiable (respectively T-unsatisfiable) if it is satisfied by some (resp., no) interpretation in I. A satisfying interpretation for φ *models* φ. A formula φ is *valid in* T (or T*-valid*), written $\models_T \varphi$, if every model of T is a model of φ. We use $\bar{a}$ to denote the tuple $(a_1, \ldots, a_n)$, $t[\bar{x}]$ for a term that may depend on $\bar{x}$, and $t[\bar{x} \mapsto \bar{s}]$ for the corresponding term where the terms $\bar{x}$ are substituted for $\bar{s}$.

A SyGuS problem for a function f in a background theory T consists of *semantic restrictions*, or a specification, for f given by a (second-order) T-formula of the form $\exists f.\, \varphi[f]$, and *syntactic restrictions* on the definitions for f, given by a context-free grammar R. A *solution for f* is a lambda term $\lambda\bar{x}.\, e$ such that $\varphi[f \mapsto \lambda\bar{x}.\, e]$ is e is in the language generated by R and is T-valid (modulo beta-reductions). For brevity, we will drop the lambda prefix when it is clear from context. We say a specification $\exists f.\, \varphi[f]$ is *point-wise* if for each clause C in its conjunctive normal form, every application of f in $\mathbf{T}(C)$ is the same term.

A. Enumerative Syntax Guided Synthesis

Enumerative SyGuS solving produces candidate solutions for the function to synthesize via exhaustive generation of all expressions from R's language, in increasing order of expression size. We define the *size of an expression* as the number of non-nullary symbols it contains, e.g., the expression x has size 0 and the expression $\text{ite}(x \geq y, x+1, y)$ has size 3.

Example 1: Consider synthesizing a binary integer function f satisfying the specification $\exists f.\, \forall x_1 x_2.\, \psi[f, x_1, x_2]$, with a solution space defined by the context-free grammar R where:

$$\begin{aligned} \psi \quad &= \quad f(x_1, x_1) \simeq x_1 + 1 \wedge f(x_1, x_1 + 1) \simeq x_1 \\ R \quad &= \quad A \to 0 \mid 1 \mid x_1 \mid x_2 \mid A + A \mid \text{ite}(B, A, A) \mid \\ & \quad\quad\ B \to A \leq A \mid \neg B \end{aligned}$$

The specification states that f relates integers and their successors: when applied over identical integers f yields their successor; and when applied over consecutive integers it yields the first one. A solution for f is $e = \lambda x_1 x_2.\, \text{ite}(x_2 \leq x_1, x_1 + 1, x_1)$, since it satisfies the specification by returning $x_1 + 1$ when $\models_T x_2 \simeq x_1$ and x_1 when $\models_T x_2 \simeq x_1 + 1$. Moreover it is minimal since there is no smaller expression that is also a solution. Enumeration will discover this solution after considering *all* expressions of size zero: 0, 1, x, x_2; then of size one: $x_1 + x_2$, $x_1 + 1$, $\ldots$; of size two: $\text{ite}(x_1 \leq x_2, x_1, 1)$, $\ldots$; and possibly many others of size three. ●

B. Counterexample-guided inductive synthesis (CEGIS)

In enumerative CEGIS [33], counterexamples from failed candidates are used to generate *refinement lemmas*: concrete instantiations of the specification that the next candidate solutions must satisfy before being verified.

Example 2: To synthesize the function f from Example 1, say the first term in the enumeration is x. The verification of $\psi[f \mapsto x_1]$ yields the counterexample $\{x_1 \mapsto 1\}$, with which we instantiate the specification and obtain the refinement lemma $f(1,1) \simeq 2 \wedge f(1,2) \simeq 1$ (L1), which all subsequent candidate solutions must satisfy. Say we then consider the candidate solution 1. Since applying $f \mapsto 1$ does not satisfy the above equalities, we discard it. The same goes for expressions x_1, x_2, $x_1 + x_2$, and so on for sizes 1 and 2. To find a candidate expression that meets the above equalities, it is necessary to reach size three, e.g., $\text{ite}(x_2 \leq 1, 1+1, 1)$. Its verification fails, however, and the counterexample $\{x_1 \mapsto 0\}$ generates the new refinement lemma $f(0,0) \simeq 1 \wedge f(0,1) \simeq 0$ (L2). The next enumerated expression to satisfy both refinement lemmas is $\text{ite}(x_2 \leq x_1, x_1 + 1, x_1)$, which is also a solution. ●

Often the bottleneck in enumerative CEGIS is generating candidate solutions, whereas invoking the verification oracle occurs relatively infrequently, as in the above example. Due to its exhaustiveness, this approach typically only performs well when solutions have relatively small term size, typically up to 5 for most grammars like the one in the previous example.

C. Divide and conquer enumeration (D&C)

In point-wise specifications, such as in Example 1, refinement lemmas constitute a set of constraints on independent inputs, which can be seen as test points: a candidate only needs to be verified if it is correct on all test cases. The D&C algorithm implemented in EUSolver [4] leverages this view by finding partial solutions, correct on subsets of the test points, and then attempting to combine them into a larger expression correct on all points. Candidates for partial solutions, i.e., terms of type of the function-to-synthesize, and conditions, i.e., predicates, are enumerated separately.

Example 3: To synthesize the function from the previous examples, D&C also operates on a CEGIS loop. After (L1) is generated, inputs $(1,1)$ and $(1,2)$ become test points. The terms $x_1 + 1$ and x_1 are generated as solutions for the first and second points, respectively, and the predicate $x_2 \leq x_1$ is generated as a separation condition for the points, allowing a candidate $\text{ite}(x_2 \leq x_1, x_1 + 1, x_1)$ to be constructed and then verified as a solution. In contrast to CEGIS, which required enumeration up to size 3, a solution can be build by D&C from enumerated expressions that have size at most 1. ●

Combining terms and predicates into candidate solutions is done by solving a multi-label decision tree learning problem. Informally (for formal definitions we refer to standard texts such as [6]), a *decision tree* is a finite binary tree with *internal nodes* consisting of *attributes* (predicates) and with *leaf nodes*, each containing *data points* and labeled with a term corresponding to the classification of these points. A set of data points is called a *sample*. In the CEGIS case, the data points are possible inputs for the function to synthesize in the refinement lemmas. In building decision trees, attributes are chosen to separate points, while leaf nodes are labeled with terms *covering* all their data points, i.e., that term is a solution for the constraints of each data point in the leaf. Finding a solution amounts to *classifying* the sample correctly, i.e., to split the sample with attributes so that every leaf node can be labeled, and converting the decision tree to a conditional expression. We assume a recursive function TOITE that for a leaf returns its label and for an internal node with attribute p and children L,R returns $\text{ite}(p, \text{TOITE}(L), \text{TOITE}(R))$.

Example 4: Solution building in Example 3 is done by learning the decision tree

$$dt \quad = \begin{tikzcd} & x_2 \leq x_1 \arrow[dl, dash, "\top"'] \arrow[dr, dash, "\bot"] & \\ \{(1,1)\} \leftarrow x_1+1 & & \{(1,2)\} \leftarrow x_1 \end{tikzcd}$$

which classifies the sample $\{(1,1),(1,2)\}$ by separating its points into leafs whose labels cover these points. Then, $\text{TOITE}(dt) = \text{ite}(x_2 \leq x_1, x_1+1, x_1)$. •

In D&C the enumeration of terms and predicates continues until the sample can be classified, with the respective candidate solution then being tested on the specification. D&C is particularly effective on PBE problems, in which the specification for the function to synthesize is essentially just a conjunction of equalities of the form $f(\bar{c}) \simeq d$ where $\bar{c}, d$ are concrete constants. In these problems, piecing together a conditional expression satisfying these equalities amounts to solving the entire synthesis problem. The main limitation of D&C is requiring specifications to be point-wise. Other specifications yield points that cannot be labeled independently, which significantly complicates the classification process.

Example 5: Consider again the SyGuS problem from Example 1, but with a specification augmented with the clause $f(x_1,x_2) \simeq x_1+1 \Rightarrow f(x_1+2,x_2) \simeq x_1$. This relational constraint makes this specification non-point-wise by introducing a dependency between different inputs of f. For example, testing a candidate solution x_1 yields the counterexample $\{x_1 \mapsto 1, x_2 \mapsto 0\}$, from which we generate the refinement lemma $f(1,1) \simeq 2 \wedge f(1,0) \simeq 2 \Rightarrow f(3,0) \simeq 1 \wedge f(1,2) \simeq 1$ (L3). Determining a solution for $(1,0)$ restricts the solution for $(3,0)$: e.g., if the former is x_1+1 then the latter must be 1, x_2+1 or another term whose output on $(3,0)$ is 1. Since D&C cannot reason about such restrictions, it cannot be applied to this problem. Moreover, a minimal solution for f, e.g., $\text{ite}(x_1 \leq x_2, \text{ite}(x_2 \leq x_1, x_1+1, x_1), x_2)$, has size 5, making this seemingly simple problem hard for enumerative CEGIS. •

III. Synthesis via Piecewise-Independent Unification (Unif+PI)

We introduce Unif+PI, a new procedure for enumerative SyGuS based, like D&C, on a CEGIS loop for the derivation of refinement lemmas and on extracting data points from these lemmas to guide the unification of partial solutions by means of decision tree learning. In contrast to D&C, however, and similarly to CEGIS in general, it can be applied to any kind of specification. Handling general, as opposed to only piece-wise, specifications introduces a serious challenge to unification: determining a solution to a given point cannot in general be done independently of other points, as shown in Example 5.

The overall architecture of Unif+PI is shown in Figure 1. Its main component is an *SMT-based classifier*. This module

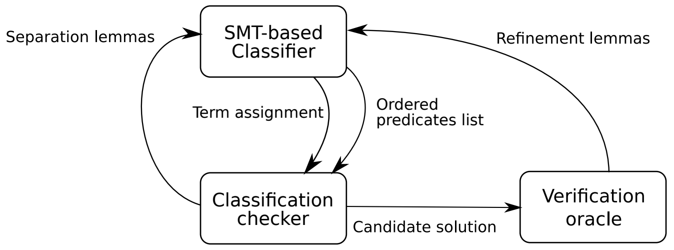


Fig. 1: Unif+PI loop.

is an instance of an SMT solver, and manages two kinds of constraints: *refinement lemmas*, as introduced earlier, and *separation lemmas*, which we describe in the following. The SMT-based classifier produces a *term assignment*: a mapping from data points, represented by their respective function applications, to terms, so that assigning each point to its respective term satisfies the refinement lemma. An example of a term assignment consistent with (L3) from Example 5 is $f(1,1) \mapsto x_2+x_2, \{f(1,0), f(3,0), f(1,2)\} \mapsto x_1$. The SMT-based classifier also generates an *ordered list of predicates* $p_1, \ldots, p_n$ whose goal is to *separate* points that are assigned to different terms. A predicate separates two points if it evaluates differently on them, e.g., $x_1 \neq x_2$ separates $(1,1)$ and $(3,0)$.

The term assignment and the ordered list of predicates correspond to a *candidate classifier*. The predicate list represents a full binary tree whose internal nodes at level i are the i-th predicate, and the term assignment represents how leafs should be labeled. The *classification checker* determines if a decision tree that correctly classifies the sample can be built from such a candidate. Since the term assignment is consistent with the refinement lemmas, the remaining condition for a correct classification is that all data points in a leaf are assigned the same term. In that case a candidate solution is extracted (with TOITE) from the decision tree. Otherwise a *separation lemma*, which explains why classification failed, is generated to force the SMT-based classifier to produce new, distinct candidate classifiers. Together, these components comprise a learner of classifiers. Their interplay is detailed in the following section.

Our procedure relies on a solution-complete search strategy which generates classifiers in increasing order of *complexity*, defined by a measure that accounts for both the size of the terms, the predicates involved, *and* their quantity. The strategy considers increasingly complex candidates until a successful classifier is learned, in which case the corresponding candidate solution is verified against the specification. Counterexamples to the candidate solution generate new refinement lemmas and cause the learning process to restart. For Example 5, whose solution is an expression of size 5, Unif+PI is able to assemble it by unifying partial solutions of size up to 1 using predicates of size 1. Our implementation of Unif+PI in the CVC4SY solver finds this solution for f in 1s, while CVC4SY's CEGIS takes

18s and EUSolver, which cannot apply its D&C and is forced to fall back to CEGIS, takes 4m20s.

A. SMT-based decision tree learning

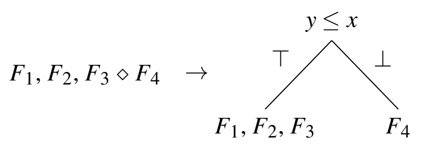
The SMT-based classifier (SMTClassify) and the classification checker (ClassChecker) are described at a high level in Figures 2a and 2b. For the enumeration of terms and predicates, SMTClassify relies on an underlying SMT solver. It encodes syntax restrictions as logical constraints whose model values correspond to expressions generated by the grammar, exploring the search space exhaustively [27, 28]. Thus φ_R and φ_S are constraints in terms of metavariables corresponding to data points and ranging over expressions to be enumerated. For instance, (L3) from Example 5 is normalized as $F_1 \simeq 2 \wedge F_2 \simeq 2 \Rightarrow F_3 \simeq 1 \wedge F_4 \simeq 1$, with metavariable F_1 corresponding to point $f(1,1)$ and so on. Separation lemmas also depend on metavariables, with P_i denoting the i-th predicate in the classifier.

Conditions (1a)-(1c) in SMTClassify refer to generating terms and predicates within the current threshold, given by n. Condition (1a) requires the number of distinct terms assigned to be at most n. This ensures a *minimally diverse* term assignment, considering first partial solutions that cover more data points. The number of predicates, $n-1$, is bound by the number of distinct terms. The rationale is that, in building a classifier, one should not need more predicates than distinct partial solutions. Conditions (1b) and (1c) require that no term or predicate have a bigger size than $\log_2(n)$, thus biasing the search towards many small expressions. These conditions guarantee that we only try a new combination of maximum term size, maximum predicate size, number of distinct terms and of distinct predicates once all candidate classifiers in the current threshold fail. Conditions (1d) and (1e) require consistency with the input refinement and separation lemmas.

Once SMTClassify produces a candidate classifier consistent with the lemmas, ClassChecker is invoked to determine whether it correctly classifies the sample. This is based on whether the sample is *separable*. This is *not* the case when two data points have the same evaluation vectors in the ordered predicate list, computed in Step (3), but are assigned different terms. This test is made in Step (4). If the sample is separable, a candidate solution is built as a conditional expression corresponding to the classifier. Otherwise, a separation lemma is generated, in Step (5). The lemma can be seen as a *trace* of incrementally building a decision tree, with predicates added to resolve *separation conflicts*: points assigned different terms having the same classification in the current tree. We explain the structure of the separation lemma in detail in Example 6 below, showing a complete execution of Unif+PI. Intuitively, it is structured so that the i-th predicate solves the i-th separation conflict. This conflict is witnessed, after the $(i-1)$-th conflict, by (a) E_i, a series of points with the same classification, (b) a pair of points $(F_{s_i}, F_{s'_i})$ assigned different terms and (c) the value of the i-th predicate separating $(F_{s_i}, F_{s'_i})$.

Example 6: Consider again the SyGuS problem from Example 5. Initially the set of refinement lemmas is empty, so no data points have been collected. The enumeration in SMTClassify produces a single term of size 0 as the candidate classifier, which is trivially a candidate solution returned by ClassChecker. Say that solution is $f = \lambda x_1 x_2 . x_1$; its verification will fail with, say, the counterexample $\{x_1 \mapsto 1, x_2 \mapsto 0\}$ which yields the refinement lemma $f(1,1) \simeq 2 \wedge f(1,0) \simeq 2 \Rightarrow f(3,0) \simeq 1 \wedge f(1,2) \simeq 1$, normalized for SMTClassify as $F_1 \simeq 2 \wedge F_2 \simeq 2 \Rightarrow F_3 \simeq 1 \wedge F_4 \simeq 1$. (To simplify presentation we ambiguously refer to data points by their metavariables F_i.) SMTClassify starts searching for a term assignment with a single distinct term of size 0 that is consistent with φ_R. No such assignment exists, which forces an increase of the enumeration threshold. After the expansion, term assignments now contain up to two distinct terms and one predicate, with maximum expression size 1. The next candidate classifier generated within this threshold that satisfies the refinement lemma is $F_1 \mapsto x_2 + x_2, \{F_2, F_3, F_4\} \mapsto x_1, P_1 \mapsto \top$. (Again to simplify presentation we represent the ordered predicate list as a mapping of indexed predicate metavariables.)

We illustrate the checking of candidate classifiers by ClassChecker via the incremental building of a decision tree as represented in a separation lemma. As soon as we consider the points corresponding to F_1 and F_2, which have different term assignments, we have a separation conflict. Since $\top$ cannot separate $(1,1)$ and $(1,0)$, the classification fails and a separation lemma $P_1 \simeq \top \Rightarrow F_1 \simeq F_2$ is produced: if the first predicate is $\top$, then F_1 and F_2 must be assigned the same term, otherwise we would have the same conflict. Adding this lemma to SMTClassify forces new candidate classifiers to either equate F_1 to F_2 in the assignment or to provide a new predicate. The next candidate classifier to satisfy the constraints is $\{F_1, F_2, F_3\} \mapsto x_2 + 1$, $F_4 \mapsto 1$, $P_1 \mapsto x_2 \leq x_1$. Running ClassChecker on it goes as follows:



	Point	$y \leq x$
F_1	$(1,1)$	$\top$
F_2	$(1,0)$	$\top$
F_3	$(3,0)$	$\top$
F_4	$(1,2)$	$\bot$

The first three points can be added to the same leaf since they are assigned the same term. A separation conflict, represented by $\diamond$, occurs when F_4 is added; however, it can be resolved by the predicate P_1 which separates the offending points. Thus the sample is classified, so the candidate solution $\mathrm{ite}(x_2 \leq x_1, x_2 + 1, 1)$ can be extracted from the decision tree. The verification fails again though, and a refinement lemma $f(-1,-1) \simeq 0 \wedge f(-1,-1) \simeq 0 \Rightarrow f(1,-1) \simeq -1 \wedge f(-1,0) \simeq -1$, nor-

SMTClassify $\longleftarrow$ $(\varphi_{\rm R}, \varphi_{\rm S}, n)$

1) If there exists $tassign = \{f(\bar{v}_i) \mapsto t_i \mid F_i \in \mathbf{T}(\varphi_{\rm R})\}$ and $preds = [p_1, \ldots, p_{n-1}]$ such that:
   a) $\#\{t \mid f(\bar{v}) \mapsto t \in tassign\} \leq n$
   b) $size(t) \leq \log_2(n)$, for each $f(\bar{v}) \mapsto t \in tassign$
   c) $size(p) \leq \log_2(n)$, for each $p \in preds$
   d) $\models_T \varphi_{\rm R}\sigma_F$ where $\sigma_F = \{F_i \mapsto (\lambda\bar{x}.t_i)\,\bar{v}_i \mid f(\bar{v}_i) \mapsto t_i \in tassign\}$
   e) $\models_T \varphi_{\rm S}\sigma_F\sigma_P$ where $\sigma_P = \{P_i \mapsto \lambda\bar{x}.\,p_i \mid 1 \leq i \leq n-1\}$

   Then, return classifier (*tassign*, *preds*)
2) Otherwise, return SMTClassify$(\varphi_{\rm R}, \varphi_{\rm S}, n+1)$

Fig. 2a: Classifier parameterized by sets of refinement lemmas $\varphi_{\rm R}$, separation lemmas $\varphi_{\rm S}$, and threshold n, with $n = 1$ initially.

ClassChecker $\longleftarrow$ $(\{f(\bar{v}_1) \mapsto t_1, \ldots, f(\bar{v}_n) \mapsto t_n\}, [p_1, \ldots, p_m])$

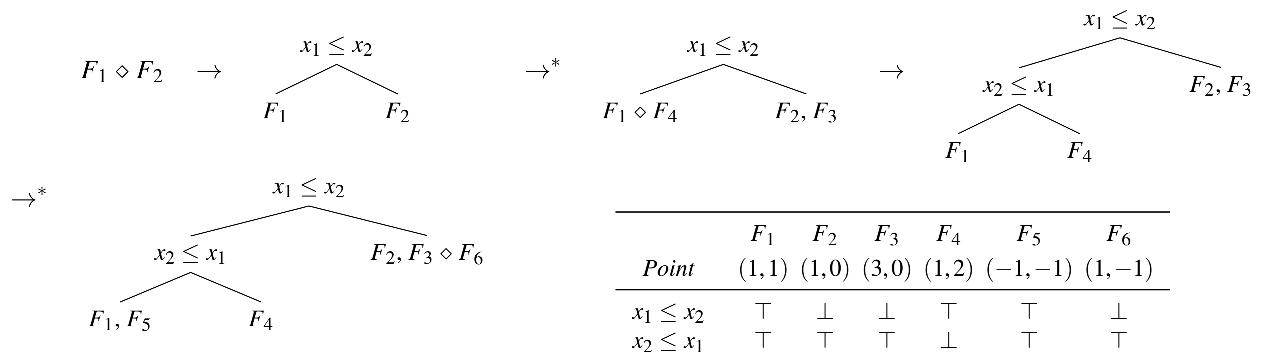
3) For a given $f(\bar{v}) \mapsto t$, let $eval(\bar{v}) = [\models_T p_1(\bar{v}), \ldots, \models_T p_m(\bar{v})]$
4) If there are no $f(\bar{v}_i) \mapsto t_i, f(\bar{v}_j) \mapsto t_j \in tassign$, for $1 \leq i < j \leq n$, such that $eval(\bar{v}_i) = eval(\bar{v}_j)$ and $\not\models_T t_i \simeq t_j$

   Then, return candidate solution TOITE$(tassign, preds)$
5) Otherwise, return separation lemma

$$E_1 \wedge F_{s_1} \not\simeq F_{s'_1} \wedge P_1 \simeq p_1 \wedge \cdots \wedge E_m \wedge F_{s_m} \not\simeq F_{s'_m} \wedge P_m \simeq p_m \Rightarrow F_i \simeq F_j$$

in which $s'_1 < \cdots < s'_m < j$, $E_1 = F_1 \simeq \cdots \simeq F_{s_1 - 1}$, and, for $l > 1$, $E_l = \bigwedge F_{k_1} \simeq F_{k_2}$, such that $eval(\bar{v}_{k_1}) = eval(\bar{v}_{k_2})$, $\models_T t_{k_1} \simeq t_{k_2}$, $k_1 < k_2$, $k_1 \leq s_l$ or $k_1 \leq s'_l$, and $k_2 > s_l, s'_l$.

Fig. 2b: Checks if a candidate classifier, given by a term assignment and an ordered list of predicates, can correctly classify sample. We assume a natural extension of TOITE for candidate classifiers.

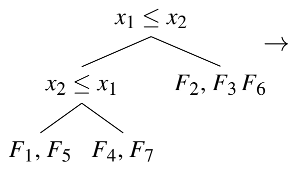


	F_1	F_2	F_3	F_4	F_5	F_6
Point	$(1,1)$	$(1,0)$	$(3,0)$	$(1,2)$	$(-1,-1)$	$(1,-1)$
$x_1 \leq x_2$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$
$x_2 \leq x_1$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$

$$SL \;=\; \underbrace{F_1 \not\simeq F_2 \wedge P_1 \simeq x_1 \leq x_2}_{\textit{1st conflict resolution}} \wedge F_2 \simeq F_3 \wedge \underbrace{F_1 \not\simeq F_4 \wedge P_2 \simeq x_2 \leq x_1}_{\textit{2nd conflict resolution}} \wedge F_1 \simeq F_5 \;\rightarrow\; \underbrace{F_3 \simeq F_6}_{\textit{Unresolved conflict}}$$

Fig. 3: Step-wise computation of separation lemma for failed classification checking.

malized as $F_5 \simeq 0 \wedge F_5 \simeq 0 \Rightarrow F_6 \simeq -1 \wedge F_7 \simeq -1$, is added to SMTClassify. A new candidate classifier satisfying the new set of lemmas can only be generated after increasing the threshold, now allowing 3 distinct terms and 2 predicates to be used, with size up to 1. The next candidate classifier provided to ClassChecker is $\{F_1, F_5\} \mapsto x_1 + 1$, $\{F_6\} \mapsto x_2$, $\{F_2, F_3, F_4\, F_7\} \mapsto x_1$, $P_1 \mapsto x_1 \leq x_2$, $P_2 \mapsto x_2 \leq x_1$, whose failed checking and resulting separation lemma are seen in Figure 3. Note that three separation conflicts occur, first $F_1 \diamond F_2$, which is resolved by P_1, then $F_1 \diamond F_4$, resolved by P_2, and finally $F_3 \diamond F_6$, which cannot be resolved. Changing the term assignment by equating F_3 (and therefore also F_2) to F_6 avoids this scenario and is consistent with the previous lemmas. With SMTClassify returning the candidate classifier $\{F_1, F_5\} \mapsto x_1 + 1$, $\{F_2, F_3, F_6\} \mapsto x_2$, $\{F_4, F_7\} \mapsto x_1$, $P_1 \mapsto x_1 \leq x_2$, $P_2 \mapsto x_2 \leq x_1$, ClassChecker verifies it and produces the minimal solution from Example 5:



$$\rightarrow \lambda x_1 x_2.\, \mathrm{ite}(x_1 \leq x_2,\ \mathrm{ite}(x_2 \leq x_1, x_1 + 1, x_1),\ x_2\ )$$ ●

B. *Unif+PI Correctness Properties*

Unif+PI satisfies the following properties where $\varphi_{\rm R}$ is the input set of refinement lemmas before normalization.

Theorem 1 (Soundness): The candidate solution produced by ClassChecker satisfies $\varphi_{\rm R}$.

Theorem 2 (Solution Completeness): If there exists a candidate solution e satisfying $\varphi_{\rm R}$, then SMTClassify$(\varphi_{\rm R}, \varphi_{\rm S}, 1)$, in which $\varphi_{\rm R}$ has been normalized and $\varphi_{\rm S}$ is initially empty but augmented after each failed execution of ClassChecker, generates a term assignment *tassign* and an ordered predicate list *preds* such that ClassChecker(*tassign*,*preds*) returns e. Moreover, e is minimal with respect to the complexity measure used by SMTClassify.

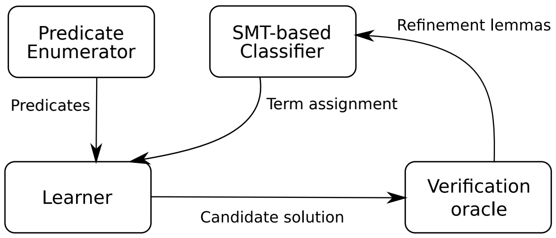


Fig. 4a: Unif+PI+E loop, with predicate enumeration independent from term enumeration.

```
LEARN ⟵ $(\{f(\bar{v}_1) \mapsto t_1, \dots, f(\bar{v}_n) \mapsto t_n\}, \{p_1, \dots, p_m\})$
  1) If $t_1 = t_2 = \cdots = t_n$ then return $t_1$
  2) If $\{p_1, \dots, p_m\} = \varnothing$ then return $\bot$
  3) Let $p$ be a heuristically selected from $\{p_1, \dots, p_m\}$ and
      a) $tassign_\top = \{f(\bar{v}_i) \mapsto t_i \mid \models_T p(\bar{v}_i)\}$, for $1 \leq i \leq n$
      b) $tassign_\bot = \{f(\bar{v}_i) \mapsto t_i \mid \not\models_T p(\bar{v}_i)\}$, for $1 \leq i \leq n$
      c) $preds = \{p_1, \dots, p_m\} \setminus \{p\}$
    Return $\text{ite}(p, \text{LEARN}(tassign_\top, preds), \text{LEARN}(tassign_\bot, preds))$
```

Fig. 4b: Decision tree learning algorithm based on ID3.

IV. Unif+PI with unconstrained predicate enumeration

The generation of separation lemmas by ClassChecker and their usage in SMTClassify, besides being instrumental for (bounded) solution completeness, biases Unif+PI to produce candidate solutions minimal with respect to expression size, number of distinct terms, and number of predicates. Sometimes, however, sacrificing completeness and minimality allows problems to be solved more efficiently. We propose a variation of Unif+PI, which we denominate Unif+PI+E, that enumerates predicates independently of previous classification attempts. The algorithm's overall architecture can be seen in Figure 4a. SMTClassify is simplified to produce only term assignments, based solely on refinement lemmas, i.e., it drops conditions (1c) and (1e). An independent predicate enumerator is responsible for generating predicates in increasing order of size, without further constraints. In this setting, more predicates are available to the learner to solve separation conflicts and successfully classify the sample. In return, the completeness and minimality guarantees of Unif+PI are lost.

The learning algorithm (Learn) of Unif+PI+E is shown in Figure 4b. It is very similar to the classical ID3 multi-label decision tree learning algorithm [24]. It takes as input a set of predicates and a term assignment and returns a conditional expression corresponding to the learned classifier. Due to the dependency between data points, it does not interleave the labeling of leafs with choosing attributes, as in ID3. Instead, the labeling is fixed by the term assignment built by SMTClassify. Step 1 checks whether the input points are assigned the same term, in which case the sample is correctly classified, corresponding to a leaf in the decision tree. Otherwise, a predicate is used to split the sample and recursively attempt to classify it (Step 3), adding an internal node to the decision tree. Classification fails when there is a separation conflict that cannot be solved (Step 2).

Learn is actually parameterized by a selection heuristic for choosing predicates in Step 3. This choice influences not only the size of the classifier, as in ID3, but also whether classification succeeds, since the labeling is fixed. We follow previous work on using extensions of information gain heuristics to function and invariant synthesis [4, 14]. Note, however, that because of our fixed labeling, we do not need to compute information gain at the level of the refinement lemmas, as done in the ICE framework for invariant synthesis [14].

The price we pay for a simpler enumeration is losing the tight integration between predicates and term assignments given by the separation lemmas in Unif+PI. Separation conflicts in Unif+PI+E are only resolved by enumerating new predicates, which can make the procedure diverge, since term assignments are not being considered exhaustively.

Another issue is that the large number of predicates available for classification makes Unif+PI+E heavily dependent on the predicate selection heuristic, as it influences not only whether a decision tree can be learned, and its size, but also which predicates compose the decision tree. This is problematic when a predicate that might be crucial for building the actual solution for the function to synthesize, not only a classifier for the sample, is ignored because other predicates that do not lead to a successful solution are selected.

V. Experiments

We implemented our approach in cvc4sy [25]. We evaluated[1] its two flavors in comparison with cvc4sy's enumerative CEGIS and with loopinvgen [22], the winner of the invariant track in SyGuS-COMP 2018 [3]. For invariant synthesis problems, we also compared our implementation with the automated invariant discovery feature of the Kind 2 model checker [8, 18]. We refer to cvc4sy's configurations by:

- cvc+c: cvc4sy's enumerative CEGIS (previous work)
- cvc+upi: cvc4sy's Unif+PI (from Section III)
- cvc+upi+e: cvc4sy's Unif+PI+E (from Section IV)

The benchmark suite from the SyGuS-COMP 2018 contains three kinds of benchmarks: PBE benchmarks, general function synthesis benchmarks, and invariant synthesis benchmarks. We do not consider PBE nor single invocation [27] benchmarks since specialized procedures [2, 4, 27] are better suited for these restricted fragments. Further eliminating benchmarks whose syntax restrictions forbid conditional expressions leaves

[1]Full data at http://homepage.divms.uiowa.edu/~hbarbosa/papers/unifpi

	Solved	Unique	Total time	Fastest	Shortest
CVC+C	**341**	30	436251	245	259
CVC+UPI+E	332	**47**	**414356**	**306**	222
CVC+UPI	291	3	494534	236	231
LOOPINVGEN	298	7	433273	261	**289**
CVC-PORT	400	-	31476	379	306

Fig. 5a: Results for 567 invariant SyGuS benchmarks, 1800s timeout. Total time is in seconds. "Fastest" (resp. "Shortest") counts how often a solution was reported among the fastest (resp. shortest) with respect to other configurations. The criteria for computing "fastest" and "shortest" are as in SyGuS-COMP 2018 [3, Section 4].

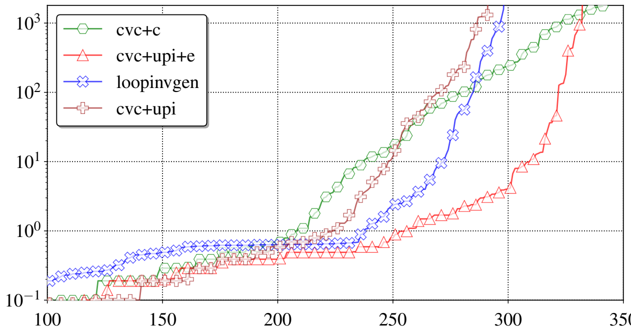


Fig. 5b: Solved problems per time for each configuration.

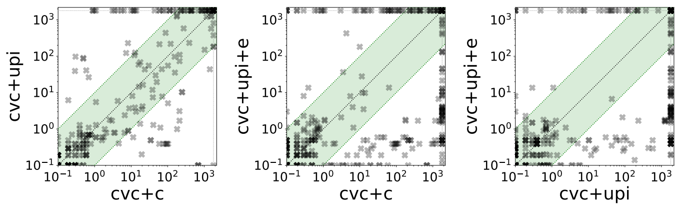


Fig. 6a: Solving time comparisons for CVC4SY configurations.

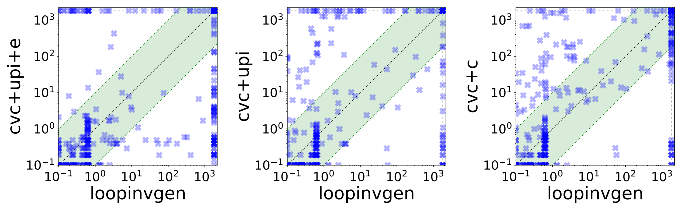


Fig. 6b: Solving time comparisons for CVC4SY vs LOOPINVGEN.

only 16 general function synthesis benchmarks for consideration. We therefore focus our evaluation on the remaining 127 invariant synthesis benchmarks and on an extra 440 invariant synthesis benchmarks stemming from the verification of Lustre [17] programs in the standard test suite of Kind 2.

A. Comparison against other SyGuS solvers

Results are summarized in Figure 5a. CVC+C solves the most problems, slightly ahead of CVC+UPI+E, while CVC+UPI and Kind 2's LOOPINVGEN have comparable performance. However, as seen in Figure 5b, CVC+UPI+E solves problems much faster than CVC+C, except for some hard problems (>1000s), and CVC+UPI is faster or comparable on problems requiring up to 100s. Moreover, as indicated in Figure 6a, in comparison with CVC+C problems solved by both CVC+UPI is on average 1.5× faster. Even more notably, CVC+UPI+E is over 100× faster. We attribute this to solutions to many invariant synthesis problems involving Boolean combinations of small literals, where divide and conquer techniques like Unif+PI are more efficient. On the other hand, CVC+C excels when invariants are only expressible by single literals with large term sizes, where divide and conquer is not helpful.

Overall, CVC+UPI+E has compelling advantages over the other solvers: Nit solves almost as many problems as CVC+C; has the greatest number of uniquely solved problems; and is often orders of magnitude faster on commonly solved problems, as can be seen in the plots in Figures 6a and 6b. A disadvantage is that the synthesized invariants it produces are generally bigger than those produced by LOOPINVGEN and other CVC4SY configurations, which seek minimal solutions. Often, CVC+UPI+E can find invariants that combine several simple attributes, while a smaller solution could be built by using fewer, more complex, attributes or by considering different term assignments for the data points. Furthermore, CVC+UPI+E may fail to find solutions that CVC+C and CVC+UPI eventually find thanks to their solution-completeness. guarantees.

The performances of CVC+UPI and LOOPINVGEN are comparable, with CVC+UPI solving 29 problems not solved by LOOPINVGEN, and LOOPINVGEN solving 36 not solved by CVC+UPI. However, it is overall slower on commonly solved problems with respect to both LOOPINVGEN and CVC+UPI+E. We believe this is due to the fact that it exhaustively explores all solutions up to a threshold, which recall from Section III-A is related to the number of predicates used in the classification and their size.

Given the different strengths of both CVC+UPI and CVC+UPI+E with respect to CVC+C, a virtual CVC4SY portfolio (CVC-PORT) including all configurations is close to the best of all worlds. As shown in Figure 5a, it solves a total of 400 problems (100 more than LOOPINVGEN), is significantly faster than all configurations taken independently, and produces shortest solutions at a similar rate as CVC+C and CVC+UPI.

B. Comparison against a model checker

To put the performance of our SyGuS approaches into perspective, we include a comparison with the Kind 2 model checker on its own set of benchmarks. In particular, this evaluation shows how our general-purpose function synthesis compares with specialized state-of-the-art model checking techniques. While the only invariant synthesis specific technique CVC4SY applies is post-condition strengthening, model checking techniques heavily exploit reachability analysis.

Kind 2 employs three induction-based model checking techniques: k-induction [30], IC3 [7], and generation of auxiliary invariants [18]. The three techniques can be run in parallel and complement each other. Kind 2 solves all the 480 benchmarks it its test suite in less than 120 seconds. When only IC3 or k-induction is enabled (without auxiliary invariant generation), the number of solved instances decreases to 445 and 313, respectively. Increasing the timeout to 1800s does not help these techniques. In contrast, no CVC4SY configuration in isolation can solve as many problems. However, with a 1800s timeout, CVC+UPI+E can solve 8 problems that IC3 cannot and 71 problems that k-induction cannot. More interestingly, if we consider CVC-PORT, it solves more problems than k-induction alone, 323 vs. 313, although is on average 10× slower. Furthermore, CVC-PORT solves 82 problems that k-induction cannot solve, and k-induction solves 72 problems that CVC-PORT cannot solve. We consider these results encouraging, as our framework is general and can be used with any theory for which a ground decision procedure is available. This is not the case for IC3, where a key element of the technique, the generalization of counterexamples, is highly dependent on the theory. Nevertheless, the effectiveness of Kind 2 shows ways of improving CVC4SY's techniques for syntax-guided invariant synthesis. Both IC3 and k-induction are greatly helped using auxiliary invariants, which is a clear direction for improving CVC4SY. A more challenging direction is to adapt IC3 techniques to the enumerative SyGuS setting.

VI. Related Work

A myriad of approaches have been developed to tackle syntax-guided synthesis, generally as instances of CEGIS but with varying ways of generating candidate solutions, such as via enumerative [33], stochastic [29], symbolic [12, 26] search or combinations thereof [2, 11, 19]. Our approach extends to general function synthesis divide and conquer techniques [4, 21]. Conceptually, however, it is closer to invariant synthesis techniques such as Padhi et al.'s precondition inference with attribute learning (PIE) [23], which is the bases for LOOPINVGEN, and Garg et al.'s ICE framework [13]. Both reduce invariant synthesis to learning a classifier for a sample built from unrollings of the transition system and from counterexamples for failed candidates. Differently from general function synthesis, however, data points are always only labeled with $\top, \bot$, which allows them to operate on a significantly simpler setting. In PIE, samples contain only positive and negative points, so classification hinges on having attributes to resolve separation conflicts. Attributes are generated via an enumerator, so their overall algorithm is similar to Unif+PI+E. Candidate invariants are generated via standard *provably approximately correct* (PAC), which is biased towards small solutions, explaining why LOOPINVGEN excels in finding concise invariants.

In ICE, samples contain implication counterexamples, so that classification must account for the dependency between these points. Since that is the only dependency in samples, its focus is on extending information-gain heuristics to cope with implications, rather than on producing a correct by construction labeling, as in our approach, before building a classifier. Recently, ICE has been extended to hHorn clause solving by Ezudheen et al. [10]. Similarly to our setting, they need to cope with complex constraints relating data points. Their decision tree learner therefore employs a Horn solver to check that their labeling does not violate the constraints.

The SMT-based classifier in Unif+PI has similarities with recent work by Narodytska et al. [20] on learning optimal decision trees with SAT solvers. They fully reduce the task of learning a minimal decision tree to a SAT problem and show they can deal with publicly available datasets of practical interest. Their encoding, however, tackles a significantly simpler problem than we do, since they consider a fixed set of attributes, only label points with $\top, \bot$, and have no dependencies between points.

VII. Conclusions and future work

We have presented Unif+PI, a new algorithm for enumerative syntax-guided synthesis based on learning classifiers using an SMT solver. It generalizes divide and conquer enumeration for non-point-wise specifications, thus extending the approach to general purpose function synthesis. We also presented Unif+PI+E, a variant that sacrifices coverage and conciseness of solutions for better performance. We implemented both algorithms in CVC4SY and extensively evaluated them on standard benchmarks sets from the SyGuS and model checking domains. Our results show that both flavors of our algorithm lead to gains in comparison with the state of the art.

In future work we will focus on refining Unif+PI+E, which shows more promising results. To improve convergence, we may address the issue of not relating term assignments and separation conflicts in a similar way to Horn-ICE [10] by adopting partial term assignments. These would be extended with suitable terms during decision tree learning, relying on an underlying SMT solver to check that the assignment still conforms with the refinement lemmas. Another direction, which would also benefit Unif+PI, is to optimize enumeration, thus improving scalability, by determining that parts of the data points are *irrelevant*. Consider, e.g., a separation conflict $f(0,0,0,1,2,1,0) \diamond f(1,0,0,5,2,1,3)$, whose resolution can be done with predicates ranging only over f's first, fourth and seventh inputs. This would reduce noise in the data by ignoring portions of counterexamples, used to generate data points, that are not necessary to witness solution verification failures.

Acknowledgments: This work was partially funded by the National Science Foundation under Award 1656926 and by DARPA under award N66001-18-C-4006.

References

[1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, 2013.

[2] R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2015.

[3] R. Alur, D. Fisman, S. Padhi, R. Singh, and A. Udupa. Sygus-comp 2018: Results and analysis. *CoRR*, abs/1904.07146, 2019.

[4] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In A. Legay and T. Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[5] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.

[9] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 2 edition, 2001.

[10] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan. Horn-ice learning for synthesizing invariants and contracts. *PACMPL*, 2(OOPSLA):131:1–131:25, 2018.

[11] G. Fedyukovich and R. Bodík. Accelerating syntax-guided invariant synthesis. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 251–269, Cham, 2018. Springer International Publishing.

[12] G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but effective functional synthesis. In C. Enea and R. Piskac, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 11388 of *Lecture Notes in Computer Science*, pages 92–113. Springer, 2019.

[13] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[14] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In R. Bodík and R. Majumdar, editors, *Symposium on Principles of Programming Languages*, pages 499–512. ACM, 2016.

[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *Symposium on Principles of Programming Languages*, pages 317–330. ACM, 2011.

[16] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pages 9–14. Springer, 2016.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[18] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2011.

[19] W. Lee, K. Heo, R. Alur, and M. Naik. Accelerating search-based program synthesis using learned probabilistic models. In J. S. Foster and D. Grossman, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 436–449. ACM, 2018.

[20] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva. Learning optimal decision trees with SAT. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1362–1368. ijcai.org, 2018.

[21] D. Neider, S. Saha, and P. Madhusudan. Compositional synthesis of piece-wise functions by learning classifiers. *ACM Trans. Comput. Log.*, 19(2):10:1–10:23, 2018.

[22] S. Padhi and T. D. Millstein. Data-driven loop invariant inference with automatic feature synthesis. *CoRR*, abs/1707.02029, 2017.

[23] S. Padhi, R. Sharma, and T. D. Millstein. Data-driven precondition inference with learned features. In C. Krintz and E. Berger, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 42–56. ACM, 2016.

[24] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.

[25] A. Reynolds, H. Barbosa, A. Nötzil, C. Barrett, and C. Tinelli. CVC4Sy: Smart and fast term enumeration for syntax-guided synthesis. In I. Dilig and S. Tasiran, editors, *Computer Aided Verification (CAV) - 31st International Conference*, (Accepted for publication). Lecture Notes in Computer Science. Springer, 2019.

[26] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2015.

[27] A. Reynolds, V. Kuncak, C. Tinelli, C. Barrett, and M. Deters. Refutation-based synthesis in smt. *Formal Methods in System Design*, 2017.

[28] A. Reynolds, A. Viswanathan, H. Barbosa, C. Tinelli, and C. Barrett. Datatypes with shared selectors. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 591–608. Springer, 2018.

[29] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316. ACM, 2013.

[30] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. H. Jr. and S. D. Johnson, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[31] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In V. Sarkar and M. W. Hall, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 281–294. ACM, 2005.

[32] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM, 2006.

[33] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In H. Boehm and C. Flanagan, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296. ACM, 2013.