Scaling Enumerative Program Synthesis via Divide and Conquer [⋆]

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa[⋆⋆]

University of Pennsylvania

Abstract. Given a semantic constraint specified by a logical formula, and a syntactic constraint specified by a context-free grammar, the Syntax-Guided Synthesis (SyGuS) problem is to find an expression that satisfies both the syntactic and semantic constraints. An enumerative approach to solve this problem is to systematically generate all expressions from the syntactic space with some pruning, and has proved to be surprisingly competitive in the newly started competition of SyGuS solvers. It performs well on small to medium sized benchmarks, produces succinct expressions, and has the ability to generalize from input-output examples. However, its performance degrades drastically with the size of the smallest solution. To overcome this limitation, in this paper we propose an alternative approach to solve SyGuS instances.

The key idea is to employ a divide-and-conquer approach by separately enumerating (a) smaller expressions that are correct on subsets of inputs, and (b) predicates that distinguish these subsets. These expressions and predicates are then combined using decision trees to obtain an expression that is correct on all inputs. We view the problem of combining expressions and predicates as a multi-label decision tree learning problem. We propose a novel technique of associating a probability distribution over the set of labels that a sample can be labeled with. This enables us to use standard information-gain based heuristics to learn compact decision trees.

We report a prototype implementation EUSOLVER. Our tool is able to match the running times and the succinctness of solutions of both standard enumerative solver and the latest white-box solvers on most benchmarks from the SyGuS competition. In the 2016 edition of the SyGuS competition, EUSOLVER placed first in the general track and the programming-by-examples track, and placed second in the linear integer arithmetic track.

1 Introduction

The field of program synthesis relates to automated techniques that attempt to automatically generate programs from requirements that a programmer writes. It has been applied to various domains such as program completion [21], program optimization, and automatic generation of programs from input-output examples [7], among others. Recently, Syntax-Guided Synthesis (SyGuS) has been

[⋆] This research was supported by NSF Expeditions award CCF 1138996 and by the Simons Investigator award.

[⋆⋆] This work was done when the author was a student at the University of Pennsylvania. The author is currently employed at Microsoft Corp.

proposed as a back-end exchange format and enabling technology for program synthesis [2]. The aim is to allow experts from different domains to model their synthesis problems as SyGuS instances, and leverage general purpose SyGuS solvers.

In the SyGuS approach, a synthesis task is specified using restrictions on both the form (syntax) and function (semantics) of the program to be synthesized: (a) The syntactic restrictions are given in terms of a context-free grammar from which a solution program may be drawn. (b) The semantic restrictions are encoded into a specification as an SMT formula. Most SyGuS solvers operate in two cooperating phases: a *learning phase* in which a candidate program is proposed, and a *verification phase* in which the proposal is checked against the specification. SyGuS solvers can be broadly categorized into two kinds: (a) black-box solvers, where the learning phase does not deal with the specification directly, but learns from constraints on how a potential solution should behave on sample inputs points [2, 18, 23]; and (b) white-box solvers, which attempt learn directly from the specification, generally using constraint solving techniques [3, 17].

The enumerative solver [2] placed first and second in the SyGuS competition 2014 and 2015, respectively. It maintains a set of concrete input points, and in each iteration attempts to produce an expression that is correct on these concrete inputs. It does so by enumerating expressions from the grammar and checking if they are correct on the input points, while pruning away expressions that behave equivalently to already generated expressions. If an expression that is correct on the input points is found, it is verified against the full specification. If it is incorrect, a counter-example point is found and added to the set of input points.

Though the enumerative strategy works well when the solutions have small sizes, it does not scale well. The time take to explore all potential solutions up to a given size grows exponentially with the size. To overcome this scalability issue, we introduce a divide-and-conquer enumerative algorithm.

The divide-and-conquer enumerative approach is based on this insight: while the full solution expression to the synthesis problem may be large, the important individual parts are small. The individual parts we refer to here are: (a) *terms* which serve as the return value for the solution, and (b) *predicates* which serve as the conditionals that choose which term is the actual return value for a given input. For example, in the expression `if` $x \leq y$ `then` y `else` x, the terms are x and y, and the predicate is $x \leq y$. In this example, although the full expression has size 6, the individual terms have size 1 each, and the predicate has size 3. Hence, the divide-and-conquer enumerative approach only enumerates terms and predicates separately and attempts to combine them into a conditional expression.

To combine the different parts of a solution into a conditional expression, we use the technique of learning decision trees [4, 16]. The input points maintained by the enumerative algorithm serve as the samples, the predicates enumerated serve as the attributes, and the terms serve as the labels. A term t is a valid label for a point `pt` if t is correct for `pt`. We use a simple multi-label decision tree learning algorithm to learn a decision tree that classifies the samples soundly, *i.e.*, for each point, following the edges corresponding to the attribute values (*i.e.*, predicates) leads to a label (*i.e.*, term) which is correct for the point.

Round no.	Enumerated expressions	Candidate Expression	Point added
1	0	0	$\{x \mapsto 1, y \mapsto 0\}$
2	0, 1	1	$\{x \mapsto 0, y \mapsto 2\}$
3	0, 1, x, y, …, $x + y$,	$x + y$	$\{x \mapsto 1, y \mapsto 2\}$
		…	
n	0, …, `if` $x \leq y$ `then` y `else` x	`if` $x \leq y$ `then` y `else` x	

Table 1: Example run of the basic enumerative algorithm

To enhance the quality of the solutions obtained, we extend the basic divide-and-conquer algorithm to be an *anytime* algorithm, *i.e.*, the algorithm does not stop when the first solution is found, and instead continues enumerating terms and predicates in an attempt to produce more compact solutions. Decomposing the verification queries into *branch-level queries* helps in faster convergence.

Evaluation. We implemented the proposed algorithm in a tool EUSOLVER and evaluated it on benchmarks from the SyGuS competition. The tool was able to perform on par or better than existing solvers in most tracks of the 2016 SyGuS competition, placing first in the general and programming-by-example tracks, and second in the linear-integer-arithmetic track. In the general and linear-integer-arithmetic tracks, EUSOLVER's performance is comparable to the state-of-the-art solvers. However, in the programming-by-example track, EUSOLVER performs exceptionally well, solving 787 of the 852 benchmarks, while no other tool solved more than 39. This exceptional performance is due to EUSOLVER being able to generalize from examples like other enumerative approaches, while also being able to scale to larger solution sizes due to the divide-and-conquer approach.

Further, to test the anytime extension, we run EUSOLVER on 50 ICFP benchmarks with and without the extension. Note that no previous solver has been able to solve these ICFP benchmarks. We observed that the anytime extension of the algorithm was able to produce more compact solutions in 18 cases.

2 Illustrative Example

Consider a synthesis task to generate an expression e such that: (a) e is generated by the grammar from Figure 1. (b) e when substituted for f, in the specification Φ, renders it true, where $\Phi \equiv \forall x, y : f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y)$. Note that the specification constrains $f(x, y)$ to return maximum of x and y. Here, the smallest solution expression is `if` $x \leq y$ `then` y `else` x.

```
S ::= T | if (C) then T else T
T ::= 0 | 1 | x | y | T + T
C ::= T ≤ T | C ∧ C | ¬ C
```

Fig. 1: Grammar for linear integer expressions

Basic Enumerative Strategy. We explain the basic enumerative algorithm [23] using Table 1. The enumerative algorithm maintains a set of input points pts (initially empty), and proceeds in rounds. In each round, it proposes a candidate solution that is correct on all of pts. If this candidate is correct on all inputs, it is returned. Otherwise, a counter-example input point is added to pts.

The algorithm generates the candidate solution expression by enumerating expressions generated by the grammar in order of size. In the first round, the candidate expression proposed is the first expression generated (the expression 0) as pts is empty. Attempting to verify the correctness of this expression, yields

Round no.	Enumerated Terms	Enumerated Predicates	Candidate Expression	Point added
1	0	0	$\emptyset$	$\{x \mapsto 1, y \mapsto 0\}$
2	0, 1	1	$\emptyset$	$\{x \mapsto 0, y \mapsto 2\}$
3	0, 1, x, y	$0 \leq 0, \ldots 0 \leq y,$ $1 \leq 0, \ldots 1 \leq y$	`if` $1 \leq y$ `then` y `else` 1	$\{x \mapsto 2, y \mapsto 0\}$
4	0, 1, x, y	$0 \leq 0, \ldots x \leq y$	`if` $x \leq y$ `then` y `else` x	

Table 2: Example run of the divide-and-conquer enumerative algorithm

a counter-example point $\{x \mapsto 1, y \mapsto 0\}$. In the second round, the expression 0 is incorrect on the point, and the next expression to be correct on all of pts (the expression 1) is proposed. This fails to verify as well, and yields the counter-example point $\{x \mapsto 0, y \mapsto 2\}$. In the third round, all expressions of size 1 are incorrect on at least one point in pts, and the algorithm moves on to enumerate larger expressions. After several rounds, the algorithm eventually generates the expression `if` $x \leq y$ `then` y `else` x which the SMT solver verifies to be correct. In the full run, the basic enumerative strategy (algorithm presented in Section 3.1) generates a large number (in this case, hundreds) of expressions before generating the correct expression. In general, the number of generated expressions grows exponentially with the size of the smallest correct expression. Thus, the enumerative solver fails to scale to large solution sizes.

Divide and Conquer Enumeration. In the above example, though the solution is large, the individual components (terms x and y, and predicate $x \leq y$) are rather small and can be quickly enumerated. The divide-and-conquer approach enumerates terms and predicates separately, and attempts to combine them into a conditional expression. We explain this idea using an example (see Table 2).

Similar to the basic algorithm, the divide-and-conquer algorithm maintains a set of points pts, and works in rounds. The first two rounds are similar to the run of the basic algorithm. In contrast to the basic algorithm, the enumeration stops in the third round after 0, 1, x, and y are enumerated – the terms 1 and y are correct on $\{x \mapsto 1, y \mapsto 0\}$ and $\{x \mapsto 0, y \mapsto 2\}$, respectively, and thus together "cover" all of pts. Now, to propose an expression, the algorithm starts enumerating predicates until it finds a sufficient number of predicates to generate a conditional expression using the previously enumerated terms. The terms and predicates are combined into conditional expression by learning decision trees (see Section 4.2). The candidate expression proposed in the third round is `if` $1 \leq y$ `then` y `else` x and the counter-example generated is $\{x \mapsto 2, y \mapsto 0\}$ (see table). Proceeding further, in the fourth round, the correct expression is generated. Note that this approach only generates 4 terms and 11 predicates in contrast to the basic approach which generates hundreds of expressions.

3 Problem Statement and Background

Let us fix the function to be synthesized f and its formal parameters params. We write range(f) to denote the range of f. The term *point* denotes a valuation of params, i.e., a point is an input to f.

Example 1. For the running example in this section, we consider a function to be synthesized f of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ with the formal parameters $\mathsf{params} = \{x, y\}$. Points are valuations of x and y. For example, $\{x \mapsto 1, y \mapsto 2\}$ is a point.

Specifications. SMT formulae have become the standard formalism for specifying semantic constraints for synthesis. In this paper, we fix an arbitrary theory $\mathcal{T}$ and denote by $\mathcal{T}[\mathsf{symbols}]$, the set of $\mathcal{T}$ terms over the set of symbols $\mathsf{symbols}$. A *specification* Φ is a logical formula in a theory $\mathcal{T}$ over standard theory symbols and the function to be synthesized f. An expression e *satisfies* Φ ($e \models \Phi$) if instantiating the function to be synthesized f by e makes Φ valid.

Example 2. Continuing the running example, we define a specification $\Phi \equiv \forall x, y : f(x, y) \geq x \wedge f(x, y) \geq y \wedge f(x, y) = x \vee f(x, y) = y$. The specification states that f maps each pair x and y to a value that is at least as great as each of them and equal to one of them, i.e., the maximum of x and y.

Grammars. An *expression grammar* G is a tuple $\langle \mathcal{N}, S, \mathcal{R} \rangle$ where: (a) the set $\mathcal{N}$ is a set of non-terminal symbols, (b) the non-terminal $S \in \mathcal{N}$ is the initial non-terminal, (c) $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{T}[\mathcal{N} \cup \mathsf{params}]$ is a finite set of rewrite rules that map $\mathcal{N}$ to $\mathcal{T}$-expressions over non-terminals and formal parameters. We say that an expression e *rewrites to* an incomplete expression e' (written as $e \to_G e'$) if there exists a rule $R = (N, e'') \in \mathcal{R}$ and e' is obtained by replacing one occurrence of N in e by e''. Let $\to_G^*$ be the transitive closure of $\to$. We say that an expression $e \in \mathcal{T}[\mathsf{params}]$ is *generated* by the grammar G (written as $e \in [\![G]\!]$) if $S \to_G^* e$. Note that we implicitly assume that all terms generated by the grammar have the right type, i.e., are of the type $\mathsf{range}(f)$.

Example 3. For the running example, we choose the following grammar. The set of non-terminals is given by $\mathcal{N} = \{S, T, C\}$ with the initial non-terminal being S. The rules of this grammar are $\{(S, T), (S, \texttt{if } C \texttt{ then } S \texttt{ else } S)\} \cup \{(T, x), (T, y), (T, 1), (T, 0), (T, T + T)\} \cup \{(C, T \leq T), (C, C \wedge C), (C, \neg C)\}$. This is the standard linear integer arithmetic grammar used for many SyGuS problems. This grammar is equivalent to the one from Figure 1.

The Syntax-Guided Synthesis Problem. An instance of the SyGuS problem is given by a pair $\langle \Phi, G \rangle$ of specification and grammar. An expression e is a solution to the instance if $e \models \Phi$ and $e \in [\![G]\!]$.

Example 4. Continuing the running example, for the specification Φ from Example 2 and the grammar from Example 3, one of the solution expressions is given by $f(x, y) \equiv \texttt{if } x \leq y \texttt{ then } y \texttt{ else } x$.

From our definitions, it is clear that we restrict ourselves to a version of the SyGuS problem where there is exactly one unknown function to be synthesized, and the grammar does not contain `let` rules. Further, we assume that our specifications are *point-wise*. Intuitively, a specification is point-wise, if it only relates an input point to its output, and not the outputs of different inputs.

Here, we use a simple syntactic notion of point-wise specifications, which we call *plain separability*, for convenience. However, our techniques can be generalized to any notion of point-wise specifications. Formally, we say that a specification is *plainly separable* if it can be rewritten into a conjunctive normal form where each clause is either (a) a tautology, or (b) each appearing application of the function to be synthesized f has the same arguments.

Example 5. The specification for our running example $\Phi \equiv f(x,y) \geq x \wedge f(x,y) \geq y \wedge f(x,y) = x \vee f(x,y) = y$ is plainly separable. For example, this implies that the value of $f(1,2)$ can be chosen irrespective of the value of f on any other point. On the other hand, a specification such as $f(x,y) = 1 \Rightarrow f(x+1,y) = 1$ is neither plainly separable nor point-wise. The value of $f(1,2)$ cannot be chosen independently of the value of $f(0,2)$.

The above restrictions make the SyGuS problem significantly easier. However, a large fraction of problems do fall into this class. Several previous works address this class of problem (see, for example, [3, 13, 17]).

Plainly separable specifications allow us to define the notion of an expression e satisfying a specification Φ on a point pt. Formally, we say that $e \models \Phi \downharpoonleft \mathsf{pt}$ if e satisfies the specification obtained by replacing each clause C in Φ by $Pre_C(\mathsf{pt}) \Rightarrow C$. Here, the premise $Pre_C(\mathsf{pt})$ is given by $\bigwedge_{p \in \mathsf{params}} Arg_C(p) = \mathsf{pt}[p]$ where $Arg_C(p)$ is the actual argument corresponding to the formal parameter p in the unique invocation of f that occurs in C. We extend this definition to sets of points as follows: $e \models \Phi \downharpoonleft \mathsf{pts} \Leftrightarrow \bigwedge_{\mathsf{pt} \in \mathsf{pts}} e \models \Phi \downharpoonleft \mathsf{pt}$.

Example 6. For the specification Φ of the running example, the function given by $f(x,y) \equiv x + y$ is correct on the point $\{x \mapsto 0, y \mapsto 3\}$ and incorrect on the point $\{x \mapsto 1, y \mapsto 2\}$

3.1 The Enumerative Solver

Algorithm 1 Enumerative Solver

```
Require: Grammar $G = \langle \mathcal{N}, S, \mathcal{R} \rangle$
Require: Specification $\Phi$
Ensure: $e$ s.t. $e \in [\![G]\!] \wedge e \models \Phi$
1: $\mathsf{pts} \leftarrow \emptyset$
2: while true do
3:     for $e \in \textsc{enumerate}(G, \mathsf{pts})$ do
4:         if $e \not\models \Phi \downharpoonleft \mathsf{pts}$ then continue
5:         $\mathsf{cexpt} \leftarrow \mathsf{verify}(e, \Phi)$
6:         if $\mathsf{cexpt} = \bot$ then return $e$
7:         $\mathsf{pts} \leftarrow \mathsf{pts} \cup \mathsf{cexpt}$
```

The principal idea behind the enumerative solver is to enumerate all expressions from the given syntax with some pruning. Only expressions that are distinct with respect to a set of concrete input points are enumerated.

The full pseudo-code is given in Algorithm 1. Initially, the set of points is set to be empty at line 1. In each iteration, the algorithm calls the ENUMERATE procedure[1] which returns the next element from a (possibly infinite) list of expressions such that no two expressions in this list evaluate to the same values at every point $\mathsf{pt} \in \mathsf{pts}$ (line 3). Every expression

[1] Note that ENUMERATE is a coprocedure. Unfamiliar readers may assume that each call to ENUMERATE returns the next expression from an infinite list of expressions.

e in this list is then verified, first on the set of points (line 4) and then fully (line 5). If the expression e is correct, it is returned (line 6). Otherwise, we pick a counter-example input point (*i.e.*, an input on which e is incorrect) and add it to the set of points and repeat (line 7). A full description of the ENUMERATE procedure can be found in [2] and [23].

Theorem 1. *Given a* SyGuS *instance* (Φ, G) *with at least one solution expression, Algorithm 1 terminates and returns the smallest solution expression.*

Features and Limitations. The enumerative algorithm performs surprisingly well, considering its simplicity, on small to medium sized benchmarks (see [2, 23]). Further, due to the guarantee of Theorem 1 that the enumerative approach produces small solutions, the algorithm is capable of generalizing from specifications that are input-output examples. However, enumeration quickly fails to scale with growing size of solutions. The time necessary for the enumerative solver to generate all expressions up to a given size grows exponentially with the size.

4 The Divide-and-Conquer Enumeration Algorithm

Conditional Expression Grammars. We introduce conditional expression grammars that separate an expression grammar into two grammars that generate: (a) the return value expression, and (b) the conditionals that decide which return value is chosen. These generated return values (terms) and conditionals (predicates) are combined using if-then-else conditional operators.

A *conditional expression grammar* is a pair of grammars $\langle G_T, G_P \rangle$ where: (a) the *term grammar* G_T is an expression grammar generating terms of type $\mathsf{range}(f)$; and (b) the *predicate grammar* G_P is an expression grammar generating boolean terms. The set of expressions $[\![\langle G_T, G_P \rangle]\!]$ generated by $\langle G_T, G_P \rangle$ is the smallest set of expressions $\mathcal{T}[\mathsf{params}]$ such that: (a) $[\![G_T]\!] \subseteq [\![\langle G_T, G_P \rangle]\!]$, and (b) $e_1, e_2 \in [\![\langle G_T, G_P \rangle]\!] \wedge p \in [\![G_P]\!] \implies \texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \in [\![\langle G_T, G_P \rangle]\!]$. Most commonly occurring SyGuS grammars in practice can be rewritten as conditional expression grammars automatically.

Example 7. The grammar from Example 3 is easily decomposed into a conditional expression grammar $\langle G_T, G_P \rangle$ where: (a) the term grammar G_T contains only the non-terminal T, and the rules for rewriting T. (b) the predicate grammar G_P contains the two non-terminals $\{T, C\}$ and the associated rules.

Decision Trees. We use the concept of decision trees from machine learning literature to model conditional expressions. Informally, a decision tree DT maps *samples* to *labels*. Each internal node in a decision tree contains an *attribute* which may either hold or not for each sample, and each leaf node contains a label. In our setting, labels are terms, attributes are predicates, and samples are points.

To compute the label for a given point, we follow a path from the root of the decision tree to a leaf, taking the left (resp. right) child at each internal node if the attribute holds (resp. does not hold) for the sample. The required label is the label at the leaf. We do not formally define decision trees, but instead refer the reader to a standard text-book (see, for example, [4]).

Example 8. Figure 2 contains a decision tree in our setting, i.e., with attributes being predicates and labels being terms. To compute the associated label with the point $\mathsf{pt} \equiv \{x \mapsto 2, y \mapsto 0\}$: (a) we examine the predicate at the root node, i.e., $y \leq 0$ and follow the left child as the predicate hold for pt; (b) examine the predicate at the left child of the root node, i.e, $x \leq y$ and follow the right child as it does not hold; and (c) return the label of the leaf $x + y$.

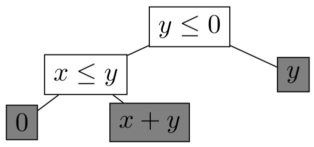


Fig. 2: Sample decision tree

The expression $\mathsf{expr}(DT)$ corresponding to a decision tree DT is defined as: (a) the label of the root node if the tree is a single leaf node; and (b) `if` p `then` $\mathsf{expr}(DT_L)$ `else` $\mathsf{expr}(DT_Y)$ where p is the attribute of the root node, and DT_L and DT_Y are the left and right children, otherwise.

Decision tree learning is a technique that learns a decision tree from a given set of samples. A decision tree learning procedure is given: (a) a set of samples (points), (b) a set of labels (terms), along with a function that maps a label to the subset of samples which it covers; and (c) a set of attributes (predicates). A sound decision tree learning algorithm returns a decision tree DT that classifies the points correctly, i.e., for every sample pt, the label associated with it by the decision tree covers the point. We use the notation LearnDT to denote a generic, sound decision tree learning procedure. The exact procedure we use for decision tree learning is presented in Section 4.2.

4.1 Algorithm

Algorithm 2 presents the full divide-and-conquer enumeration algorithm for synthesis. Like Algorithm 1, the divide-and-conquer algorithm maintains a set of points pts, and in each iteration: (a) computes a candidate solution expression e (lines 3-10); (b) verifies and returns e if it is correct (lines 10 and 11); and (c) otherwise, adds the counter-example point into the set pts (line 12).

However, the key differences between Algorithm 2 and Algorithm 1 are in the way the candidate solution expression e is generated. The generation of candidate expressions is accomplished in two steps.

Term solving. Instead of searching for a single candidate expression that is correct on all points in pts, Algorithm 2 maintains a set of candidate terms terms. We say that a term t covers a point $\mathsf{pt} \in \mathsf{pts}$ if $t \models \Phi \downharpoonleft \mathsf{pt}$. The set of points that a term covers is computed and stored in $\mathsf{cover}[t]$ (line 15). Note that the algorithm does not store terms that cover the same set of points as already generated terms (line 16). When the set of terms terms covers all the points in pts, i.e., for each $\mathsf{pt} \in \mathsf{pts}$, there is at least one term that is correct on pt, the term enumeration is stopped (while-loop condition in line 4).

Unification and Decision Tree Learning. In the next step (lines 6-9), we generate a set of predicates preds that will be used as conditionals to combine the terms from terms into if-then-else expressions. In each iteration, we attempt to learn a decision tree that correctly labels each point $\mathsf{pt} \in \mathsf{pts}$ with a term t such that $\mathsf{pt} \in \mathsf{cover}[t]$. If such a decision tree DT exists, the conditional expression

$\mathsf{expr}(DT)$ is correct on all points, i.e., $\mathsf{expr}(DT) \models \Phi \downarrow \mathsf{pts}$. If a decision tree does not exist, we generate additional terms and predicates and retry.

Algorithm 2 DCSolve: The divide-and-conquer enumeration algorithm

```
Require: Conditional expression grammar $G = \langle G_T, G_P \rangle$
Require: Specification $\Phi$
Ensure: Expression $e$ s.t. $e \in [\![G]\!] \wedge e \models \Phi$
 1: $\mathsf{pts} \leftarrow \emptyset$
 2: while true do
 3:     $\mathsf{terms} \leftarrow \emptyset$; $\mathsf{preds} \leftarrow \emptyset$; $\mathsf{cover} \leftarrow \emptyset$; $DT = \bot$
 4:     while $\bigcup_{t \in \mathsf{terms}} \mathsf{cover}[t] \neq \mathsf{pts}$ do                ▷ Term solver
 5:         $\mathsf{terms} \leftarrow \mathsf{terms} \cup$ NextDistinctTerm($\mathsf{pts}, \mathsf{terms}, \mathsf{cover}$)
 6:     while $DT = \bot$ do                                         ▷ Unifier
 7:         $\mathsf{terms} \leftarrow \mathsf{terms} \cup$ NextDistinctTerm($\mathsf{pts}, \mathsf{terms}, \mathsf{cover}$)
 8:         $\mathsf{preds} \leftarrow \mathsf{preds} \cup$ enumerate($G_P, \mathsf{pts}$)
 9:         $DT \leftarrow$ LearnDT($\mathsf{terms}, \mathsf{preds}$)
10:     $e \leftarrow \mathsf{expr}(DT)$; $\mathsf{cexpt} \leftarrow \mathsf{verify}(e, \Phi)$          ▷ Verifier
11:     if $\mathsf{cexpt} = \bot$ then return $e$
12:     $\mathsf{pts} \leftarrow \mathsf{pts} \cup \mathsf{cexpt}$
13: function NextDistinctTerm($\mathsf{pts}, \mathsf{terms}, \mathsf{cover}$)
14:     while True do
15:         $t \leftarrow$ enumerate($G_T, \mathsf{pts}$); $\mathsf{cover}[t] \leftarrow \{\mathsf{pt} \mid \mathsf{pt} \in \mathsf{pts} \wedge t \models \Phi \downarrow \mathsf{pt}\}$
16:         if $\forall t' \in \mathsf{terms} : \mathsf{cover}[t] \neq \mathsf{cover}[t']$ then return $t$
```

Remark 1. In line 7, we generate additional terms even though terms is guaranteed to contain terms that cover all points. This is required to achieve semi-completeness, i.e., without this, the algorithm might not find a solution even if one exists.

Theorem 2. *Algorithm 2 is sound for the* SyGuS *problem. Further, assuming a sound and complete* LearnDT *procedure, if there exists a solution expression, Algorithm 2 is guaranteed to find it.*

The proof of the above theorem is similar to the proof of soundness and partial-completeness for the original enumerative solver. The only additional assumption is that the LearnDT decision tree learning procedure will return a decision tree if one exists. We present such a procedure in the next section.

4.2 Decision Tree Learning

The standard multi-label decision tree learning algorithm (based on ID3 [16]) is presented in Algorithm 3. The algorithm first checks if there exists a single label (i.e., term) t that applies to all the points (line 1). If so, it returns a decision tree with only a leaf node whose label is t (line 1). Otherwise, it picks the best predicate p to split on based on some heuristic (line 3). If no predicates are left, there exists no decision tree, and the algorithm returns $\bot$ (line 2). Otherwise, it recursively computes the left and right sub-trees for the set of points on which p holds and does not hold, respectively (lines 4 and 5). The final decision tree is returned as a tree with a root (with attribute p), and positive and negative edges to the roots of the left and right sub-trees, respectively.

Algorithm 3 Learning Decision Trees

```
Require: $\mathsf{pts}, \mathsf{terms}, \mathsf{cover}, \mathsf{preds}$
Ensure: Decision tree $DT$
1: if $\exists t : \mathsf{pts} \subseteq \mathsf{cover}[t]$ then return $\mathit{LeafNode}[\mathcal{L} \leftarrow t]$
2: if $\mathsf{preds} = \emptyset$ then return $\bot$
3: $p \leftarrow$ Pick predicate from $\mathsf{preds}$
4: $L \leftarrow \textsc{LearnDT}(\{\mathsf{pt} \mid p[\mathsf{pt}]\}, \mathsf{terms}, \mathsf{cover}, \mathsf{preds} \setminus \{p\})$
5: $R \leftarrow \textsc{LearnDT}(\{\mathsf{pt} \mid \neg p[\mathsf{pt}]\}, \mathsf{terms}, \mathsf{cover}, \mathsf{preds} \setminus \{p\})$
6: return $\mathit{InternalNode}[\mathcal{A} \leftarrow p, \mathit{left} \leftarrow L, \mathit{right} \leftarrow R]$
```

Information-gain heuristic. The choice of the predicate at line 3 influences the size of the decision tree learned by Algorithm 3, and hence, in our setting, the size of the solution expression generated by Algorithm 2. We use the classical information gain heuristic to pick the predicates. Informally, the information gain heuristic treats the label as a random variable, and chooses to split on the attribute knowing whose value will reveal the most information about the label. We do not describe all aspects of computing information gain, but refer the reader to any standard textbook on machine learning [4]. Given a set of points $\mathsf{pts}' \subseteq \mathsf{pts}$ the entropy $H(\mathsf{pts}')$ is defined in terms of the probability $\mathbb{P}_{\mathsf{pts}'}(\mathit{label}(\mathsf{pt}) = t)$ of a point $\mathsf{pt} \in \mathsf{pt}'$ being labeled with the term t as

$$H(\mathsf{pts}') = -\sum_t \mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t) \cdot \log_2 \mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t)$$

Further, given a predicate $p \in \mathsf{preds}$, the information gain of p is defined as

$$G(p) = \frac{|\mathsf{pts}_\mathrm{y}|}{|\mathsf{pts}|} \cdot H(\mathsf{pts}_\mathrm{y}) + \frac{|\mathsf{pts}_\mathrm{n}|}{|\mathsf{pts}|} \cdot H(\mathsf{pts}_\mathrm{n})$$

where $\mathsf{pts}_\mathrm{y} = \{\mathsf{pt} \in \mathsf{pts} \mid p[\mathsf{pt}]\}$ and $\mathsf{pts}_\mathrm{n} = \{\mathsf{pt} \in \mathsf{pts} \mid \neg p[\mathsf{pt}]\}$. Hence, at line 3, we compute the value $G(p)$ for each predicate in preds, and pick the one which maximizes $G(p)$.

We use conditional probabilities $\mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t \mid \mathsf{pt})$ to compute the probability $\mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t)$. The assumption we make about the prior distribution is that the likelihood of a given point pt being labeled by a given term t is proportional to the number of points in $\mathsf{cover}[t]$. Formally, we define:

$$\mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t \mid \mathsf{pt}) = \begin{cases} 0 & \text{if } \mathsf{pt} \notin \mathsf{cover}[t] \\ \dfrac{|\mathsf{cover}[t] \cap \mathsf{pts}'|}{\sum\limits_{t' \mid \mathsf{pt} \in \mathsf{cover}[t']} |\mathsf{cover}[t'] \cap \mathsf{pts}'|} & \text{if } \mathsf{pt} \in \mathsf{cover}[t] \end{cases}$$

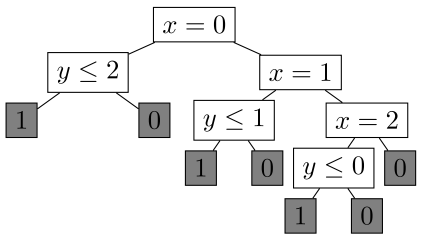
Now, the unconditional probability of an arbitrary point being labeled with t is given by $\mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t) = \sum_{\mathsf{pt}} \mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t \mid \mathsf{pt}) \cdot \mathbb{P}_{\mathsf{pts}'}(\mathsf{pt})$. Assuming a uniform distribution for picking points, we have that

$$\mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t) = \frac{1}{|\mathsf{pts}|} \cdot \sum_{\mathsf{pt}} \mathbb{P}_{\mathsf{pts}'}(\mathsf{label}(\mathsf{pt}) = t \mid \mathsf{pt})$$

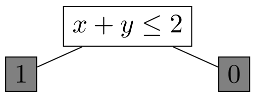
4.3 Extensions and Optimizations

The Anytime Extension. Algorithm 2 stops enumeration of terms and predicates as soon as it finds a single solution to the synthesis problem. However, there are cases where due to the lack of sufficiently good predicates, the decision tree and the resulting solution can be large (see Example 9). Instead, we can let the algorithm continue by generating more terms and predicates. This could lead to different, potentially smaller decision trees and solutions.

Example 9. Given the specification $(x \geq 0 \wedge y \geq 0) \Rightarrow (f(x,y) = 1 \Leftrightarrow x + y \leq 2)$ and a run of Algorithm 2 where the terms 0 and 1 are generated; the terms fully cover any set of points for this specification. Over a sequence of iterations the predicates are generated in order of size. Now, the predicates generated of size 3 include $x = 0$, $x = 1$, $x = 2$, $y \leq 2$, $y \leq 1$, and $y \leq 0$. With these predicates, the decision tree depicted in Figure 3a is learned, and the corresponding conditional expression is correct for the specification. However, if the procedure continues to run after the first solution is generated, predicates of size 4 are generated. Among these predicates, the predicate $x + y \leq 2$ is also generated. With this additional predicate, the decision tree in Figure 3b is generated, leading to the compact solution $f(x,y) \equiv$ `if` $x + y \leq 2$ `then` 1 `else` 0.



(a) Decision tree for predicates of size 3



(b) Decision tree for predicates of size 4

Decision Tree Repair. In Algorithm 2, we discard the terms that cover the same set of points as already generated terms in line 16. However, these discarded terms may lead to better solutions than the already generated ones.

Example 10. Consider a run of the algorithm for the running example, where the set pts contains the points $\{x \mapsto 1, y \mapsto 0\}$ and $\{x \mapsto -1, y \mapsto 0\}$. Suppose the algorithm first generates the terms 0 and 1. These terms are each correct on one of the points and are added to terms. Next, the algorithm generates the terms x and y. However, these are not added to terms as x (resp. y) is correct on exactly the same set of points as 1 (resp. 0).

Suppose the algorithm also generates the predicate $x \leq y$ and learns the decision tree corresponding to the expression $e \equiv$ `if` $x \leq y$ `then` 0 `else` 1. Now, verifying this expression produces a counter-example point, say $\{x \mapsto 1, y \mapsto 2\}$. While the term 0, and correspondingly, the expression e is incorrect on this point, the term y which was discarded as an equivalent term to 0, is correct.

Hence, for a practical implementation of the algorithm we do not discard these terms and predicates, but store them separately in a map Eq : terms $\rightarrow [\![G_T]\!]$ that maps the terms in terms to an additional set of equivalent terms. At lines 16, if the check for distinctness fails, we instead add the term t to the Eq map. Now, when the decision tree learning algorithm returns an expression that fails to verify and returns a counter-example, we attempt to replace terms and predicates in the decision tree with equivalent ones from the Eq map to make the decision tree behave correctly on the counter-example.

Example 11. Revisiting Example 10, instead of discarding the terms x and y, we store them into the Eq array, i.e., we set $Eq(0) = \{y\}$ and $Eq(1) = \{x\}$. Now, when the verification of the expression fails, with the counter-example point $\{x \mapsto 1, y \mapsto 2\}$, we check the term that is returned for the counter-example point–here, 0. Now, we check whether any term in $Eq(0)$ is correct on the counter-example point–here, the term y. If so, we replace the original term with the equivalent term that is additionally correct on the counter-example point and proceed with verification. Replacing 0 with y in the expression gives us `if` $x \leq y$ `then` y `else` 1. Another round of verification and decision tree repair will lead to replacing the term 1 with x, giving us the final correct solution.

Branch-wise verification. In Algorithm 2, and in most synthesis techniques, an incorrect candidate solution is used to generate one counter-example point. However, in the case of conditional expressions and point-wise specifications, each branch (i.e., leaf of the decision tree) can be verified separately. Verifying each branch involves rewriting the specification as in the point-wise verification defined in Section 3 – but instead of adding a premise to each clause asserting that the arguments to the function are equal to a point, we add a premise that asserts that the arguments satisfy all predicates along the path to the leaf. This gives us two separate advantages:

- We are able to generate multiple counter-examples from a single incorrect expression. This reduces the total number of iterations required, as well as the number of calls to the expensive decision tree learning algorithm.
- It reduces the complexity of each call to the verifier in terms of the size of the SMT formula to be checked. As verification procedures generally scale exponentially with respect to the size of the SMT formula, multiple simpler verification calls are often faster than one more complex call.

This optimization works very well along with the decision tree repair described above as we can verify and repair each branch of the decision tree separately.

Example 12. Consider the verification of the expression `if` $x \leq y$ `then` 0 `else` 1 for the running example. Instead of running the full expression through the verifier to obtain one counter-example point, we can verify the branches separately by checking the satisfiability of the formulae $x \leq y \wedge f(x, y) = 0 \wedge \neg\Phi$ and $\neg(x \leq y) \wedge f(x, y) = 1 \wedge \neg\Phi$. This gives us two separate counter-example points.

5 Evaluation

We built a prototype SyGuS solver named EUSOLVER that uses the divide-and-conquer enumerative algorithm. The tool consists of 6000 lines of Python code implementing the high-level enumeration and unification algorithms, and 3000 lines of C++ code implementing the decision tree learning. The code is written to be easily extensible and readable, and has not been optimized to any degree. All experiments were executed on the Starexec platform [22] where each benchmark is solved on a node with two 4-core 2.4GHz Intel processors and 256GB of RAM, with a timeout of 3600 seconds.

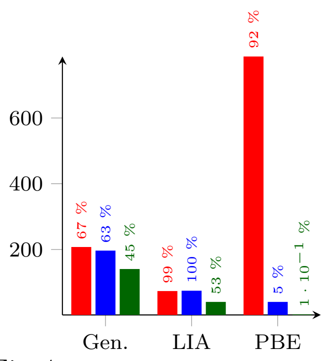


Fig. 4: Number of benchmarks solved per track for EUSOLVER (red), CVC4 (blue), and ESOLVER (green)

Goals. We seek to empirically evaluate how our synthesis algorithm compares to other state-of-the-art synthesis techniques along the following dimensions: (a) *Performance*: How quickly can the algorithms arrive at a correct solution? (b) *Quality*: How *good* are the solutions produced by the algorithms? We use compactness of solutions as a metric for the quality of solutions. (c) *Effect of anytime extension*: How significant is the improvement in the quality of the solutions generated if the algorithm is given an additional (but fixed) time budget?

Benchmarks. We draw benchmarks from 3 tracks of the SyGuS competition 2016: [2]

(a) *General track.* The general track contains 309 benchmarks drawn from a wide variety of domains and applications.

(b) *Programming-by-example track.* The PBE track contains 852 benchmarks where, for each benchmark, the semantic specification is given by a set of input-output examples.

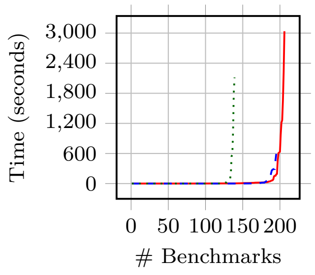
(c) *Linear-integer-arithmetic track.* The LIA track contains 73 benchmarks, each over the linear integer arithmetic theory, where the grammar is fixed to a standard grammar that generates conditional linear arithmetic expressions.

5.1 Discussion

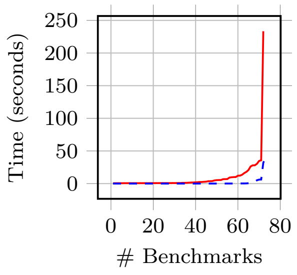
Figures 5 and 4 plot the full results of running EUSOLVER on the benchmarks from the three categories. The plots also contain the results of 2 other state-of-the-art solvers: (a) the white-box solver CVC4-1.5.1 based on [17], and (b) the enumerative black-box solvers ESOLVER described in [2]

Performance. EUSOLVER was able to solve 206 of the 309 benchmarks in the general track and 72 of the 73 benchmarks in the PBE track. CVC4 solves 195 and 73 benchmarks in these categories, while ESOLVER solves 139 and 34. As Figure 5 shows, the performance is comparable to both CVC4 and ESOLVER in both tracks, being only marginally slower in the LIA track. However, EUSOLVER performs exceptionally well on the PBE benchmarks, solving 787 while CVC4 solved 39 and ESOLVER solved 1. PBE benchmarks require the solver to generalize from input-output examples—EUSOLVER inherits this ability from the enumerative approach.

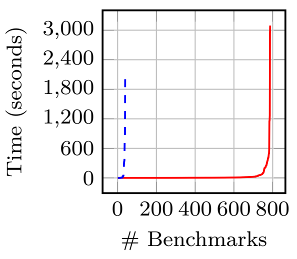
[2] The SyGuS competition 2016 included an addition track – the invariant generation track. However, the specifications in this track are not simply separable, and EUSOLVER falls back to the standard enumeration algorithm instead of the divide-and-conquer techniques described in this paper.



(a) General track.



(b) LIA track.



(c) PBE track.

Interpretation: For every point, the x-coordinate gives the number of benchmarks which are solved within the time indicated by the y-coordinate.

Fig. 5: Running times for ESOLVER (dotted), CVC4 (dashed), and EUSOLVER (solid)

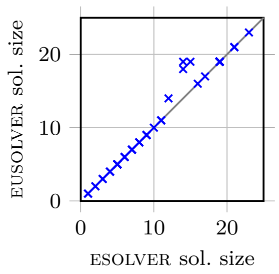


Fig. 6: Scatter plot of EUSOLVER and ESOLVER solution sizes.

However, the standard enumerative solver ESOLVER is unable to solve these benchmarks due to the large solution sizes—EUSOLVER overcomes this hurdle with the divide-and-conquer approach.

Quality of Solutions. Figure 6 highlights the solution sizes produced by EUSOLVER and ESOLVER for the commonly solved benchmarks in the general track. EUSOLVER often matches the solution sizes produced by ESOLVER (108 of the 112 benchmarks). ESOLVER is guaranteed to produce the smallest solution possible. This shows that the divide-and-conquer approach does not significantly sacrifice solution quality for better performance.

Effect of Anytime Extension. We selected 50 ICFP benchmarks from the general track and use them to test the anytime extension described in Section 4.3. The ICFP benchmarks are synthesis tasks that were first proposed as a part of the ICFP programming contest 2013, which were then adapted to the SyGuS setting. To the best of our knowledge, no other SyGuS solver has been able to solve the ICFP benchmarks satisfactorily. For 18 of the 50 ICFP benchmarks, we were able to obtain a more compact solution by letting the algorithm continue execution after the first solution was discovered. Further, the difference in the first and smallest solutions is sometimes significant—for example, in the case of the “icfp_118_100” benchmark, we see a reduction of 55%. An interesting phenomenon that we observed was that while the size of the decision tree almost always went down with time, the size of the solutions sometimes increased. This is because the algorithm generated larger terms and predicates over time, increasing the size of the labels and attributes in each node of the decision tree.

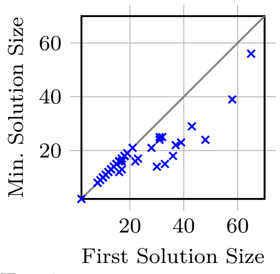


Fig. 7: Scatter plot of first vs. minimum size solutions with the anytime extension. Points below $x = y$ benefit from the anytime extension.

Overall, our experiments suggests that: (a) The DCSolve algorithm is able to quickly learn compact solutions, and generalizes well from input-output examples.

(b) The anytime nature of DCSolve often reduces the size of the computed solution;
(c) The DCSolve algorithm works competently on problems from different domains.

6 Concluding Remarks

Related Work. Program synthesis has seen a revived interest in the last decade, starting from the SKETCH framework [20, 21] which proposed counterexample guided inductive synthesis (CEGIS). Most synthesis algorithms proposed in recent literature can be viewed as an instantiation of CEGIS. Synthesis of string manipulating programs using examples has found applications in Microsoft's FlashFill [7], and the ideas have been generalized in a meta-synthesis framework called FlashMeta [15]. Other recent work in the area of program synthesis have used type-theoretic approaches [9, 14] for program completion and for generating code snippets. Synthesis of recursive programs and data structure manipulating code has also been studied extensively [1, 5, 12]. Lastly, synthesis techniques based on decision trees have been used to learn program invariants [6].

In the area of SyGuS, solvers based on enumerative search [23], stochastic search [2, 19] and symbolic search [8, 11] were among the first solvers developed. The SKETCH approach has also been used to develop SyGuS solvers [10]. Alchemist [18] is another solver that is quite competitive on benchmarks in the linear arithmetic domains. More recently, white box solvers like the CVC4 solver [17] and the unification based solver [3] have also been developed.

The enumerative synthesis algorithm used by ESOLVER [2, 23] and the work on using decision trees for piece-wise functions [13] are perhaps the most closely related to the work described in this paper. We have already discussed at length the shortcomings of ESOLVER that our algorithm overcomes. The approach for learning piece-wise functions [13] also uses decision trees. While the presented framework is generic, the authors instantiate and evaluate it only for the linear arithmetic domain with a specific grammar. In DCSolve, neither the decision tree learning algorithm, nor the enumeration is domain-specific, making DCSolve a domain and grammar agnostic algorithm. The algorithm presented in [13] can easily learn large constants in the linear integer domain. This is something that enumerative approaches, including DCSolve, struggle to do. The heuristics used for decision tree learning are different; in [13], the authors use a heuristic based on hitting sets, while we use an information gain heuristic with cover-based priors.

Conclusion. This paper has presented a new enumerative algorithm to solve instances of the Syntax-Guided Synthesis (SyGuS) problem. The algorithm overcomes the shortcomings of a basic enumerative algorithm by using enumeration to only learn small expressions which are correct on subsets of the inputs. These expressions are then used to form a conditional expression using Boolean combinations of enumerated predicates using decision trees. We have demonstrated the performance and scalability of the algorithm by evaluating it on standard benchmarks, with exceptional performance on programming-by-example benchmarks. The algorithm is generic, efficient, produces compact solutions, and is *anytime* — in that continued execution can potentially produce more compact solutions.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013*, pages 934–950, 2013.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided Synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20–23, 2013*, pages 1–8, 2013.

[3] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II*, pages 163–179, 2015.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[5] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the* 36th *ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[6] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512, 2016.

[7] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011*, pages 317–330, 2011.

[8] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011*, pages 62–73, 2011.

[9] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion using Types and Weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38, 2013.

[10] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 377–394, 2015.

[11] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010*, pages 215–224, 2010.

[12] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis Modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications,*

OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 407–426, 2013.

[13] Parthasarathy Madhusudan, Daniel Neider, and Shambwaditya Saha. Synthesizing Piece-wise Functions by Learning Classifiers. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, Netherlands, April 2 – 8, 2016. Proceedings*, 2016.

[14] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed Program Synthesis. In *Proceedings of the* 36th *ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015*, pages 619–630, 2015.

[15] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25–30, 2015*, pages 107–126, 2015.

[16] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[17] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II*, pages 198–216, 2015.

[18] Shambwaditya Saha, Pranav Garg, and P. Madhusudan. Alchemist: Learning Guarded Affine Functions. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 440–446, 2015.

[19] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA - March 16 – 20, 2013*, pages 305–316, 2013.

[20] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415.

[21] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294, 2005.

[22] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pages 367–373, 2014. URL `http://dx.doi.org/10.1007/978-3-319-08587-6_28`.

[23] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16–19, 2013*, pages 287–296, 2013.