Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis

WOOSUK LEE, Hanyang University, South Korea

We present an effective method for scalable and general-purpose inductive program synthesis. There have been two main approaches for inductive synthesis: enumerative search, which repeatedly enumerates possible candidate programs, and the top-down propagation (TDP), which recursively decomposes a given large synthesis problem into smaller subproblems. Enumerative search is generally applicable but limited in scalability, and the TDP is efficient but only works for special grammars or applications. In this paper, we synergistically combine the two approaches. We generate small program subexpressions via enumerative search and put them together into the desired program by using the TDP. Enumerative search enables to bring the power of TDP into arbitrary grammars, and the TDP helps to overcome the limited scalability of enumerative search. We apply our approach to a standard formulation, syntax-guided synthesis (SyGuS), thereby supporting a broad class of inductive synthesis problems. We have implemented our approach in a tool called Duet and evaluate it on SyGuS benchmark problems from various domains. We show that Duet achieves significant performance gains over existing general-purpose as well as domain-specific synthesizers.

CCS Concepts: • **Software and its engineering → Programming by example**; *Domain specific languages.*

Additional Key Words and Phrases: Programming by example, Syntax-guided Synthesis

ACM Reference Format:
Woosuk Lee. 2021. Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (January 2021), 28 pages. https://doi.org/10.1145/3434335

1 INTRODUCTION

Inductive program synthesis aims to generate a program in a domain-specific language (DSL) that works correctly on an inductive specification. Here, an inductive specification consists of a set of input-output examples of the form $a \mapsto b$, where a is input, and b is the output of the desired program on input a. Since input-output examples are readily available and appealing to end-users who need to perform programming tasks but lack the expertise to write code, inductive synthesis has gained increasing academic and industrial attention. The FlashFill [Gulwani 2011] feature of Microsoft Excel is one of the most well-known examples of automated end-user programming. Inductive synthesis techniques have also been applied to other domains such as circuit transformation [Eldib et al. 2016; Lee et al. 2020], super optimization [Phothilimthana et al. 2016], program repair [Mechtaev et al. 2016], among many others.

There is a dichotomy in inductive synthesis strategies: *general-purpose* strategies which work for arbitrary DSLs, and *domain-specific* strategies that are only applicable to specific kinds of DSLs. General-purpose strategies tend to be less efficient than domain-specific ones. Since synthesis problems are notoriously difficult combinatorial search problems [Caulfield et al. 2015], speeding

Author's address: Woosuk Lee, College of Computing, Hanyang University, South Korea, woosuk@hanyang.ac.kr.

2475-1421/2021/1-ART54
https://doi.org/10.1145/3434335

up synthesis often requires domain knowledge in various forms, thereby limiting the application scope within specific domains.

The enumerative search strategies are generally applicable to arbitrary DSLs, but inefficient. The strategies traverse the search space following a specific order. The simplest one is the *bottom-up enumerative search* [Alur et al. 2017; Udupa et al. 2013]. It begins with constructing small programs and then putting together progressively larger programs until a correct program is found. To prune the search space, it employs a general optimization technique using *observational equivalence* with respect to input examples. Despite the optimization, however, the enumerative strategy is difficult to scale to large programs, because the search space grows exponentially with the size of the program.

The *top-down propagation* (TDP) (also called top-down deductive search [Polozov and Gulwani 2015]) strategies – also adopted by FlashFill– are efficient but not generally applicable to arbitrary DSLs. A given synthesis problem is recursively decomposed into multiple subproblems and their solutions are combined. In particular, after hypothetically deciding an overall structure of the target program expression satisfying certain input-output examples, for example $F(e_1, \cdots, e_k)$, specialized rules are used to *deduce* new input-output examples that should be *propagated* to its subexpressions $e_1, \cdots, e_k$. Such rules for deducing specifications on subexpressions are often called *inverse semantics* operators. The TDP is not directly applicable when there are (virtually) infinitely many possible argument values that can result in the desired output. For instance, suppose we try to synthesize an expression of the form $e_1 \oplus e_2$ satisfying input-output examples where $\oplus$ denotes the 64-bit bitwise XOR operator. To deduce specifications on the subexpressions e_1 and e_2, we need to enumerate all possible 2^{64} argument values. To finitize or further narrow down the value space, the previous work adopts either of the following two approaches or both:

- *Using domain-specific knowledge about operators in the underlying DSL.* For instance, suppose we try to synthesize a list manipulating expression of kind $\texttt{map}(f, x)$ satisfying a specification $[1, 2] \mapsto [3, 4]$ where x denotes input and f is an unknown function subexpression to be synthesized. Note that the space of values for f is infinite. Considering the property of `map`, we can deduce two new input-output examples for f: $1 \mapsto 3$ and $2 \mapsto 4$. Inductive synthesizers Myth [Frankle et al. 2016] and λ^2 [Feser et al. 2015] adopt this approach.
- *Restricting the expressivity of the underlying DSL.* FlashFill adopts this approach. The FlashFill DSL grammar restricts the use of substring extraction operator `Substring`, which takes a string and two specified indices. The grammar only allows the `Substring` operator to take input examples, preventing the other operators from being nested. This syntactic restriction allows deducing two missing specified indices for a given specification by identifying the starting and ending positions of the desired output in the input example. In general, this approach requires a great deal of domain expertise to achieve *balanced expressivity* of the underlying DSL [Gulwani 2016]: the DSL should be expressive enough for practical uses while being restricted enough to enable the TDP.

Since the above both approaches require domain knowledge, the TDP has been customized for specific application domains such as string/list manipulation, data extraction [Le and Gulwani 2014], format normalization [Kini and Gulwani 2015], and program transformation [Rolim et al. 2017].

In this paper, we propose a general approach to make the TDP applicable to arbitrary DSLs, thereby enabling a more scalable general-purpose synthesis strategy. Our key observation is that while the bottom-up enumerative search is not scalable enough to find a large solution expression, it can at least quickly enumerate its small subexpressions. Based on this insight, after enumerating

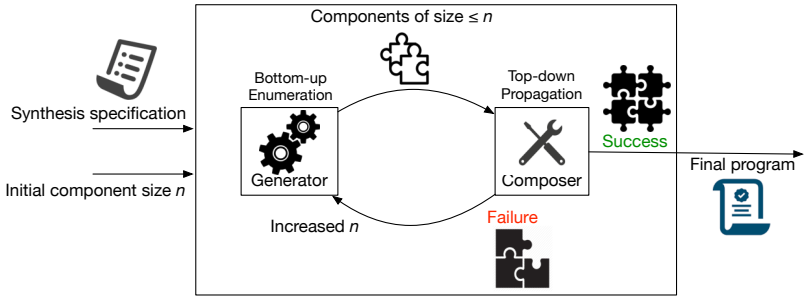


Fig. 1. High-level architecture of our synthesis algorithm.

small expressions and obtaining a set of values to which the expressions evaluate on given input examples, we confine the value space of *unconstrained* arguments to be the set.

Fig. 1 depicts the high-level architecture of our synthesis algorithm based on the synergistic combination of the TDP and bottom-up enumeration. Our algorithm consists of two phases: *component generation* and *composition*. Given an inductive synthesis problem and the initial size of component expressions n, we first construct a *component library* by generating expressions of size $\leq n$ using the bottom-up enumeration. Through the TDP, we attempt to construct a solution from the components. Our TDP repeatedly decomposes a given synthesis problem to subproblems, each of which is further split, or directly solved by using an expression in the component library. The desired program may not be synthesized if the given set of components is insufficient. Then, we augment the component library by adding larger expressions and attempt to construct a solution again. This process is repeated until a solution is found.

To make the approach feasible, we had to address two key challenges: 1) the composition phase should efficiently determine unrealizability in case of insufficient component expressions so that more component expressions can be promptly generated, 2) the number of component expressions should be controlled since it determines the computational cost of the composition phase. We address the first challenge by designing an efficient but incomplete composition algorithm; our composition algorithm may fail to construct a solution even with enough components available. This incompleteness does not harm the search completeness of the overall algorithm though because the enumerative step will eventually enumerate a solution of finite size. We address the second challenge by exploiting the existing pruning technique based on observational equivalence which reduces the number of component expressions by removing redundant expressions.

While our approach is generally applicable to any kind of language, we have applied the approach to the SyGuS [Alur et al. 2013] specification language. SyGuS is a standard formation that has established various synthesis benchmarks through annual competitions. SyGuS employs a formal grammar to describe the space of possible programs. Such a grammar is expressible in some SMT theory. We devise inverse semantics operators specialized for the operators in theories of strings, bitvectors, linear integer arithmetics (LIA), and SAT. By targetting the standard formulation, our synthesis algorithm is applicable for a broad class of SyGuS problems with arbitrary grammars in those theories.

We implemented our approach in a tool called Duet. We evaluate Duet on 1,536 benchmark problems from three widely applicable domains: string manipulation (end-user programming problems), bitvector manipulation (efficient low-level algorithms), and circuit transformation (attack-resistant crypto circuits and optimized homomorphic evaluation circuits).

We compared Duet against the state-of-the-art SyGuS solvers: EUSolver and CVC4. Duet is able to solve 1,443 problems in less than 30 seconds on average per problem, compared to

only 1,258 and 987 by EUSolver and CVC4 using 2 and 5 minutes on average, respectively. We also compare Duet against the state-of-the-art solver Euphony guided by probabilistic models. Euphony outperforms Duet in the bitvector domain, but performs worse than Duet in the other domains, and overall, Duet is 9x faster on average, solving more problems than Euphony. Duet thus provides significant performance gains that are complementary to those achieved by existing synthesizers.

We summarize the main contributions of our work:

- A novel and general approach for efficient inductive synthesis: we bring the power of the TDP to synthesis problems with arbitrary DSLs by incorporating it with the bottom-up enumerative search.
- An efficient inductive synthesis algorithm usable for a wide range of inductive synthesis problems: by targetting the SyGuS specification language, our algorithm can be used for inductive SyGuS problems with arbitrary grammars. The algorithm is guaranteed to find a correct solution if exists.
- Inverse semantics operators for theories of strings, bitvectors, LIA and SAT: the inverse semantics operators for SyGuS are not only a core part of our synthesis system but also can be a useful reference for applying our approach to languages other than the SyGuS specification language.
- Implementation and evaluation on benchmark problems from a variety of widely applicable domains: the results demonstrate significant performance gains over existing synthesis techniques.

2 OVERVIEW

We illustrate existing approaches and our method on the problem of synthesizing a string manipulating program. The desired string-manipulating program is a function f that takes as input a string of a phone number (denoted x) and outputs one in a different format by removing a leading space and a plus symbol, replacing all hyphens and spaces in the middle with dots, and adding a dot at the end. For example, given a string "␣+6␣775-969-238", the function is supposed to return "6.775.969.238.". [1]

We formulate this problem as an instance of the *syntax-guided synthesis* (SyGuS) problem [Alur et al. 2013]. Each formulation comprises a *syntactic specification*, in the form of a *regular-tree grammar* that confines the space of possible programs, and a *semantic specification*, in the form of a logical formula which defines a correctness condition of the desired program.

The syntactic specification for f is the grammar:

$$\begin{aligned} S &\rightarrow x \mid \text{“␣”} \mid \text{“-”} \mid \text{“.”} \mid \texttt{SubStr}(S, I, I) \mid \texttt{Rep}(S, S, S) \mid \texttt{ConCat}(S, S) \mid \cdots \\ I &\rightarrow 1 \mid 2 \mid I - I \mid \texttt{Length}(S) \mid \cdots \end{aligned}$$

where S is the start non-terminal symbol, and the operators are the ones supported in the theory of strings. `ConCat` and `Length` are the string concatenation/length operators, respectively, $\texttt{SubStr}(s, i, j)$ returns the substring of s that begins at index i and extends to the length $j(> 0)$, and $\texttt{Rep}(s, t_1, t_2)$ returns a new string where the first occurrence of t_1 in s is replaced by t_2.

The semantic specification for f follows the *programming by example* (PBE) paradigm and comprises input-output examples given as a logical formula:

$$f(\underbrace{\text{“␣+6␣775-969-238”}}_{i_1}) = \underbrace{\text{“6.775.969.238.”}}_{o_1} \wedge f(\underbrace{\text{“␣+82 10-56-80”}}_{i_2}) = \underbrace{\text{“82.10.56.80.”}}_{o_2}.$$

[1] A slight variant of the `phone-9.sl` benchmark used in the annual SyGuS competition.

[2] Let us denote the two input and output examples by i_1, i_2 and o_1, o_2 respectively. Each expression e in the grammar produces a vector of outputs $\langle [\![e]\!](i_1), [\![e]\!](i_2)\rangle$ on the input examples. The smallest solution that produces $\langle o_1, o_2\rangle$ is

$$\overbrace{\texttt{Rep}(\texttt{Rep}(\texttt{Rep}(\texttt{SubStr}(\underbrace{\texttt{ConCat}(x, \text{“.”})}_{\text{output: } \langle \text{“␣+6␣775-969-238.”}, \text{“␣+82␣10-56-80.”}\rangle}, 2, \texttt{Length}(x) - 1), \text{“␣”}, \text{“.”}), \text{“-”}, \text{“.”}), \text{“-”}, \text{“.”})}^{e}.$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{output: } \langle \text{“6␣775-969-238.”}, \text{“82␣10-56-80.”}\rangle}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{output: } \langle \text{“6.775.969.238.”}, \text{“82.10.56.80.”}\rangle} \tag{1}$$

The size of the solution in AST nodes is 18.

2.1 Existing Approaches

Both of the bottom-up enumerative search and the TDP-based search cannot be used to solve this problem. The solution has a large syntax-tree representation (size 18) and is difficult to be synthesized by enumeration. The bottom-up enumeration-based synthesizers search for the solution by enumerating programs generated by the given grammar in order of size. During the search, they prune the search space using *observational equivalence* with respect to a given set of input-output examples. For example, since two programs $\texttt{SubStr}(x, 0, 1)$ and “␣” generate the same output, only the smaller expression “␣” is maintained and used for constructing larger candidates. Though this optimization has been shown effective in various application domains (e.g., EUSolver adopting this strategy won the general track in 2016 SyGuS competition [Past SyGuS Competition 2020]), the solution cannot be found even within 6 hours since the number of enumerated candidates grows exponentially in program size in general.

In particular, the previous methods based on the TDP are not applicable since the grammar is not restricted enough. Based on a hypothesis about the structure of the desired program, they construct a solution candidate by recursively decomposing a synthesis problem into simpler subproblems and obtaining a solution for each one. In this example, assuming the desired solution is of the form $\texttt{Rep}(e_1, e_2, e_3)$, the problem is supposed to be decomposed into subproblems of finding subexpressions e_1, e_2 and e_3 such that e_1 evaluates to a string and e_2 (resp. e_3) evaluates to a proper match (resp. replacement) string to obtain the desired output. Since there are infinitely many possible argument values, we cannot deduce finitely many input-output examples for the subexpressions. For this reason, to the best of our knowledge, there is no known TDP-based method generally applicable to SyGuS problems with arbitrary grammars.

2.2 Our Approach

Our key observation is as follows: the bottom-up enumerative search quickly identifies small but important subexpressions of a desired program (e.g., $\texttt{ConCat}(x, \text{“.”})$ and $\texttt{Length}(x)$ in the example). Thus, we generate expressions using the bottom-up enumeration and weave them together into a solution by leveraging on the top-down propagation. Fig. 2 depicts the process of finding the solution using our method.

Component Generation. We first generate component expressions. As depicted in Fig. 1, the user may provide an integer n as the initial upperbound of sizes of component expressions. The

[2] Our approach is also applicable to a broad class of SyGuS instances beyond the cases where input-output examples are provided upfront. Please see Section 3 for more details.

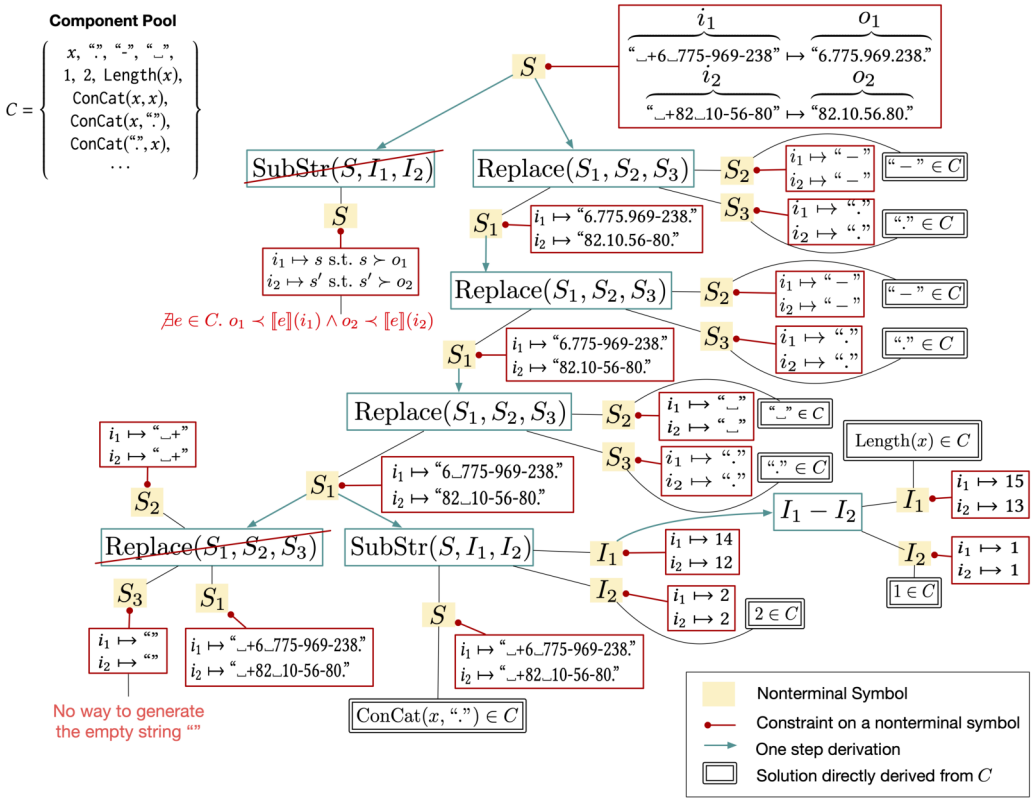


Fig. 2. Top-down propagation with component expressions generated by Bottom-up enumeration on the motivating example where $\prec$ denotes the substring relation.

larger n we use, the more chance we get for finding a solution in the composition phase at the cost of extra overhead on both the generator and the composer. The number of component expressions is potentially exponential to n, but in practice, the pruning technique based on observational equivalence significantly decreases the number of components. The component set monotonically grows in the process of our synthesis algorithm. If we fail to find a solution in the composition phase, we increase n by 1 and add larger expressions into the component set and repeat the composition phase. Suppose we are given $n = 3$. Using the bottom-up enumerative search, we get a set C of component expressions of size ≤ 3, which is shown in the left top of Fig. 2. Equipped with C, we go into the composition phase.

Composition based on the TDP. Our composition phase based on the TDP alternates between the following two steps:

(1) Given some examples for a nonterminal N (initially the start symbol S), if there exists a component expression derivable from N satisfying the examples, use it to directly solve the given synthesis problem. If not, invoke synthesis on all right-hand side (RHS) rules of N, and unite the results. Each RHS rule corresponds to a hypothesis about the structure of the target program derivable from N.
(2) Deduce examples that should be propagated to arguments $N_1, \cdots, N_k$ of a current rule $N \longrightarrow F(N_1, \cdots, N_k)$ to decompose the problem for N into multiple sub-problems.

$[\![\texttt{ConCat}(i_1, \text{“.”})]\!]$	␣	+	6	␣	7	7	5	−	9	6	9	−	2	3	8	.
o_1 (desired output)	ϵ	ϵ	6	.	7	7	5	.	9	6	9	.	2	3	8	.

$[\![\texttt{ConCat}(i_2, \text{“.”})]\!]$	␣	+	8	2	␣	1	0	−	5	6	−	8	0	.
o_2 (desired output)	ϵ	ϵ	8	2	.	1	0	.	5	6	.	8	0	.

Fig. 3. Alignments of the desired outputs and outputs produced by $\texttt{ConCat}(x, \text{“.”})$.

Suppose we first consider a hypothesis $\texttt{SubStr}(S, I_1, I_2)$. We attempt to generate examples on the first argument S. Given the input string i_1 (resp. i_2), the first argument expression should evaluate to a string "containing" o_1 (resp. o_2). Since there may be infinitely many such strings, we cannot infer finitely many input/output examples for S.

Finitizing the Value Spaces of Unconstrained Arguments by using Components. When we encounter such an argument that has infinitely many possible values (which we call an *unconstrained* argument), we limit the value space of that argument within the set D of output vectors produced by the expressions in C on the input examples (i.e., $D = \{\langle [\![e]\!](i_1), [\![e]\!](i_2)\rangle \mid e \in C\}$). If we can find an expression $e \in C$ that produces a vector of strings each of which contains the corresponding output example, then by fixing the first argument S to be e, we will be able to infer proper examples on I_1 and I_2 by identifying the starting and ending positions of o_1 (resp. o_2) in $[\![e]\!](i_1)$ (resp. $[\![e]\!](i_2)$). Unfortunately, there is no such expression in C. We reject the hypothesis and move on to other hypotheses.

By doing so, we may miss a solution of the form $\texttt{SubStr}(S, I_1, I_2)$ at this iteration because the value set derived from C does not contain the values of all possible argument expressions. For example, the following expression is such a solution:

$$\texttt{SubStr}(\underbrace{\texttt{Rep}(\texttt{Rep}(\texttt{Rep}(\texttt{Rep}(\texttt{ConCat}(x, \text{“.”}), \text{“␣”}, \text{“.”}), \text{“␣”}, \text{“.”}), \text{“-”}, \text{“.”}), \text{“-”}, \text{“.”})}_{e'}, 2, \texttt{Length}(x) - 1). \quad (2)$$

However, the overall algorithm does not compromise the search completeness. Though we could not find the solution (2) at this time, we will eventually find it if we keep increasing n until when $n = 15$ because C will contain the subexpression e' of size 15 in (2).

Next, based on another hypothesis $\texttt{Rep}(S_1, S_2, S_3)$, we attempt to generate examples on the argument expressions. Similarly to the previous case, there are infinitely many possible values for S_1, and we may let the value space of S_1 be the set D. However, to find the solution in this manner, we would need the expression e in the solution (1) of which size is still too large (15) to be generated by the component generator. To address this problem, we guide the decomposition to generate sub-problems that can be eventually solved with the current set C of small expressions.

Inverse Semantics Specialized for the SyGuS Language Constructs. We have devised an effective inverse semantics operator for the Rep function, which enables to construct the solution with the current set C. For the purpose of finding a small solution, we search for an expression in C which produces strings *similar* to o_1 and o_2 respectively so that a minimal number of string replacement operations may result in the desired output examples. The similarity is defined by string edit distance.

For each $e \in C$, we compute edit distance between $[\![e]\!](i_1)$ and o_1 (resp. $[\![e]\!](i_2)$ and o_2), and find out $\texttt{ConCat}(x, \text{“.”})$ produces strings closest to the output examples. We compute an *alignment* of $[\![\texttt{ConCat}(x, \text{“.”})]\!](i_1)$ and o_1 (resp. $[\![\texttt{ConCat}(x, \text{“.”})]\!](i_2)$ and o_2). An alignment is a pair of strings obtained by arranging two strings to identify regions of similarity as depicted in Fig. 3 where ϵ

denotes the empty word and the highlighted in color denote *mismatched* characters. From the alignments, we get a list of pairs of match/replacement strings (which we call *replacement pairs*)

$$[(\text{“␣+”}, \text{“”}), (\text{“␣”}, \text{“.”}), (\text{“-”}, \text{“.”}), (\text{“-”}, \text{“.”})] \tag{3}$$

where each element represents a contiguous mismatched segment. On $[\![\texttt{ConCat}(x, \text{“.”})]\!](i_1)$ (resp. $[\![\texttt{ConCat}(x, \text{“.”})]\!](i_2)$), we repeatedly perform string replacement using the replacement pairs *without the last pair* and obtain “6.775.969-238.” (resp. “82.10.56-80.”). Now we have found proper constraints on the argument expressions; given the input examples, the first argument should produce ⟨“6.775.969-238.”, “82.10.56-80.”⟩, and the other arguments should produce ⟨“-”, “-”⟩ and ⟨“.”, “.”⟩, respectively. We continue the deductive search only for the first argument since the others can be directly solved by component expressions “-” and “.” respectively.

We turn to the problem of finding an expression that produces ⟨“6.775.969-238.”, “82.10.56-80.”⟩. Suppose again we are based on the hypothesis $\texttt{Rep}(S_1, S_2, S_3)$. After a similar process, we find constraints on the argument expressions; given the input examples, the first argument should produce ⟨“6.775-969-238.”, “82.10-56-80.”⟩, and the others should produce ⟨“-”, “-”⟩ and ⟨“.”, “.”⟩, respectively.

Next, we find an expression that produces ⟨“6.775-969-238.”, “82.10-56-80.”⟩. Suppose again we are based on the hypothesis $\texttt{Rep}(S_1, S_2, S_3)$. After a similar process, we conclude the first argument should produce ⟨“6␣775-969-238.”, “82␣10-56-80.”⟩. The other arguments are expected to produce ⟨“␣”, “␣”⟩ and ⟨“.”, “.”⟩ which are the outputs of components “␣” and “.” respectively.

Now we try to find an expression that produces ⟨“6␣775-969-238.”, “82␣10-56-80.”⟩. Suppose again we are based on the hypothesis $\texttt{Rep}(S_1, S_2, S_3)$. The constraints on S_1, S_2 and S_3 are to produce ⟨“6␣775-969-238.”, “82␣10-56-80.”⟩, ⟨“␣+”, “␣+”⟩ and ⟨“”, “”⟩ respectively. We cannot solve the subproblem for S_3 because we do not have the empty string as a constant string in the grammar, and we cannot generate it through any of the allowed string operators – concatenation, replacement, and substring extraction. Thus we reject the hypothesis.

We move on to another hypothesis that the desired expression is of the form $\texttt{SubStr}(S, I_1, I_2)$. The constraint on S is to output a string containing “6␣775-969-238.” given i_1 (resp. “82␣10-56-80.” given i_2). The component $\texttt{ConCat}(x, \text{“.”})$ satisfies this constraint. The constraints on I_1 and I_2 are to produce $\langle 2, 2\rangle$ and $\langle 14, 12\rangle$ respectively. That is because $\texttt{SubStr}(\texttt{ConCat}(i_1, \text{“.”}), 2, 14)$ and $\texttt{SubStr}(\texttt{ConCat}(i_2, \text{“.”}), 2, 12)$ evaluate to the desired outputs o_1 and o_2 respectively. The constraint on I_1 is directly solvable by using the component expression 2. To solve the subproblem for I_2, we use a hypothesis that the solution is of the form $I_1 - I_2$. Again, there are infinitely many combinations of two expressions e_1 and e_2 such that $e_1 - e_2$ produces $\langle 14, 12\rangle$. We search for an expression in C that evaluates to an integer greater than 14 on i_1 (resp. 12 on i_2). We choose $\texttt{Length}(x)$ because it produces $\langle 15, 13\rangle$. By setting the expected output for I_1 to be $\langle 15, 13\rangle$, the supposed output of I_2 is determined to be $\langle 1, 1\rangle$ accordingly. The subproblem for I_2 is directly solvable by using the component expression 1. Now that every subproblem has been solved, the solution is found.

Using Version Space Algebra. At this point, we may stop or continue searching for a better solution by exploring other hypotheses. If the user is interested in finding a solution that (locally) maximizes an objective function, we get all solutions obtainable by this method, store them into a space-efficient data structure called *version space algebra* (VSA), which has been often used for PBE tasks [Gulwani 2011], and pick the best one.

The rest of the paper is organized as follows. Section 3 introduces preliminary concepts. Section 4 describes our algorithm of combining the top-down propagation and bottom-up enumeration. Section 5 presents our experimental results. Section 6 discusses related work and Section 7 concludes.

3 PRELIMINARIES

In this section, we introduce preliminary concepts including syntax-guided synthesis over a finite set of examples, the top-down deductive synthesis strategy and version-space algebra (some notations are borrowed from [Hu et al. 2020] and [Gulwani et al. 2017]).

Term. A signature Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative interger n, the arity of f (denoted $arity(f)$). For $n \geq 0$, we denote the set of all n-ary elements Σ by $\Sigma^{(n)}$. Let X be a set of variables. The set $T_{\Sigma,X}$ of all Σ-terms over X is inductively defined; $X \subseteq T_{\Sigma,X}$ and $\forall n \geq 0, f \in \Sigma^{(n)}.\ t_1, \cdots, t_n \in T_{\Sigma,X}.\ f(t_1, \cdots, t_n) \in T_{\Sigma,X}$.

Regular Tree Grammar. A regular tree grammar (RTG) is a tuple $G = (N, \Sigma, S, \delta)$ where N is a finite set of nonterminal symbols of arity 0; Σ is a signature; $S \in N$ is an initial nonterminal; and δ is a finite set of productions of the form $A_0 \rightarrow \sigma^{(i)}(A_1, \cdots, A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a nonterminal. Given a tree (or a term) $t \in T_{\Sigma,X}$, applying a production $r = A \rightarrow \beta$ into t is replacing the left-most occurrence of A in t with the right-hand side β. A tree $t \in T_{\Sigma,X}$ is generated by the grammar G – denoted by $t \in L(G)$ – iff it can be obtained by applying a sequence of productions $r_1, \cdots, r_n$ to the tree of which root node represents the initial nonterminal S. $\delta_A \subseteq \delta$ denotes the set of productions associated with nonterminal A. We denote $L(G)\mid_A$ as a set of trees that can be derived from the nonterminal A.

Syntax-Guided Synthesis (SyGuS). Given a background theory $\mathcal{T}$ (e.g., linear integer arithmetic) over a signature Σ, the goal of a SyGuS problem is to find a function f that satisfies a *syntactic* and a *semantic* constraints provided by the user. A syntactic constraint limits the search space of f and given as an RTG G that defines a subset of all terms in $T_{\Sigma,X}$. A semantic constraint is a logical formula defining the correctness condition of f. We denote a SyGuS problem as a pair $\langle \varphi(f(\vec{x}), \vec{x}), G\rangle$ where $\varphi(f(\vec{x}), \vec{x})$ is a semantic constraint and $\vec{x}$ denotes inputs. We will denote a fact that a function f is valid for a specification φ by $f \models \varphi$.

Most SyGuS solvers do not solve the problem of finding a term satisfying the specification on all inputs. Instead, they search for an expression satisfying the specification on a set of finite examples using Counterexample-Guided Inductive Synthesis (CEGIS) [Solar-Lezama et al. 2006]. If such a term is found, it is validated to check if it can be generalized to all inputs.

Definition 3.1. Given a SyGuS problem $sy = \langle \varphi(f(\vec{x}), \vec{x}), G\rangle$ and a finite vector of inputs $\vec{\mathbf{e}}^{in} = \langle i_1, \cdots, i_n\rangle$, let $sy^{\vec{\mathbf{e}}^{in}} := \langle \varphi^{\vec{\mathbf{e}}^{in}}(f), G\rangle$ denote the problem of finding a term $P \in L(G)$ such that $[\![P]\!]$ is only required to be correct on the examples in $\vec{\mathbf{e}}^{in}$. Let $[\![P]\!]_{\vec{\mathbf{e}}^{in}}$ denote the vector of outputs $\langle [\![P]\!](i_1), \cdots, [\![P]\!](i_n)\rangle$ produced by P on $\vec{\mathbf{e}}^{in}$. A $sy^{\vec{\mathbf{e}}^{in}}$ is *realizable* if there exists P such that $\varphi^{\vec{\mathbf{e}}^{in}} = \bigwedge_{i_j \in \vec{\mathbf{e}}^{in}} \varphi([\![P]\!](i_j), i_j)$ holds, *unrealizable* otherwise.

SyGuS-PBE. We call a SyGuS problem sy a *SyGuS-PBE* instance if there exists a finite vector of inputs $\vec{\mathbf{e}}^{in} = \langle i_1, \cdots, i_n\rangle$ such that

- the specification $\varphi^{\vec{\mathbf{e}}^{in}}$ of $sy^{\vec{\mathbf{e}}^{in}}$ can be written in the form of $\bigwedge_{i_j \in \vec{\mathbf{e}}^{in}} [\![P]\!](i_j) = o_j$ for some constants $o_1, \cdots, o_n$,
- $sy^{\vec{\mathbf{e}}^{in}}$ is realizable, and there exists a solution of $sy^{\vec{\mathbf{e}}^{in}}$ that can be generalized to sy.

Not only SyGuS problems that use a semantic specification comprising input-output examples (i.e., PBE) but also SyGuS instances having a *functional* semantic specification (i.e., there exists a single unique output satisfying the specification for the same input) also SyGuS-PBE instances. For example, SyGuS problems where semantic specifications are of the form $\forall \vec{x}.\ f(\vec{x}) = f'(\vec{x})$ for

some known function f' which appear in applications such as code optimization [Phothilimthana et al. 2016], program deobfuscation [Jha et al. 2010], and security [Eldib et al. 2016] fall into this category. In such cases, $\vec{\mathbf{e}}^{in}$ can be found through CEGIS.

Witness Functions. The top-down propagation is performed by using witness functions (also called inverse semantics operators). Given a regular tree grammar G with a signature Σ having a production rule $N_0 \to F(N_1, \cdots, N_k)$, a witness function F_j^{-1} deduces a specification φ_j on j-th argument subexpression for a given specification φ on $F(N_1, \cdots, N_k)$. The deduced specification φ_j is *necessary* iff [3]

$$\forall P_j.\ P_j \not\models \varphi_j \implies \nexists P_1, \cdots, P_{j-1}, P_{j+1}, \cdots, P_k.\ F(P_1, \cdots, P_{j-1}, P_j, P_{j+1} \cdots, P_k) \models \varphi.$$

The specification φ_j is *sufficient* iff

$$\forall P_j.\ P_j \models \varphi_j \implies \exists P_1, \cdots, P_{j-1}, P_{j+1}, \cdots, P_k.\ F(P_1, \cdots, P_{j-1}, P_j, P_{j+1} \cdots, P_k) \models \varphi.$$

Since, in general, argument subexpression are not independent of each other, all subexpression specifications should be constructed at the same time, which is costly. Therefore, *conditional witness functions* as per-argument decomposition of inverse semantics are often used.

A conditional witness function of F for j-th argument is a function $F_j^{-1}(\varphi \mid N_1 \models \varphi_1, \cdots, N_{j-1} \models \varphi_{j-1})$ that deduces a specification φ_j on N_j assuming *prerequisite* arguments $N_1, \cdots, N_{j-1}$ satisfy specifications $\varphi_1, \cdots, \varphi_{j-1}$, respectively. [4] The deduced specification φ_j is necessary iff

$$\begin{aligned}\forall P_j.\ P_j \not\models \varphi_j \implies \nexists P_1, \cdots, P_{j-1}, P_{j+1}, \cdots, P_k.\ & \\ (\forall 1 \le i < j.\ P_i \models \varphi_i) \wedge F(P_1, \cdots, P_{j-1}, P_j, P_{j+1} \cdots, P_k) \models \varphi.&\end{aligned}$$

In this paper, we only consider conditional witness functions and specifications comprising input-output examples. In addition, we often denote a disjunctive specification as a set (the empty set corresponds to false).

Version Space Algebra (VSA). Given a regular tree grammar $G = \langle N, \Sigma, S, \delta \rangle$, a version space algebra is a representation for a set $\widetilde{N}$ of programs in $L(G)\ |_N$. The grammar of VSAs is:

$$\widetilde{N} \to \{P_1, \cdots, P_k\} \mid \bigcup(\widetilde{N_1}, \cdots, \widetilde{N_k}) \mid F_{\bowtie}(\widetilde{N_1}, \cdots, \widetilde{N_k})$$

where $F \in \Sigma^{(k)}$ is any k-ary operator in $L(G)$, and P_j are some programs in $L(G)$. Intuitively, a VSA is a directed acyclic graph (DAG) where each node represents a set of programs. Leaf nodes contain explicit enumerations of programs; they are composed into larger sets by two possible VSA constructor nodes. Union nodes $\bigcup$ represent a set union of their constituent VSAs. Join nodes $F_{\bowtie}$ represent a cross-product of their constituent VSAs, with an associated operator F applied to all combinations of parameter programs from the cross-product.

The semantics of VSA as a set of programs is given as follows:

$$\begin{aligned} P \in \{P_1, \cdots, P_k\} & \qquad (\exists j : P = P_j) \\ P \in \bigcup(\widetilde{N_1}, \cdots, \widetilde{N_k}) & \qquad (\exists j : P \in \widetilde{N_j}) \\ P \in F_{\bowtie}(\widetilde{N_1}, \cdots, \widetilde{N_k}) & \qquad (P = F(P_1, \cdots, P_k) \wedge \forall j : P_j \in \widetilde{N_j}). \end{aligned}$$

The key property of VSAs is their ability to encode exponential sets of programs in polynomial space. They achieve that by providing two kinds of sharing among program subexpressions. One is

[3] All quantified variables refer to terms in $T_{\Sigma, X}$ unless otherwise specified.

[4] In the paper [Polozov and Gulwani 2015] where the concept of conditional witness functions is introduced, functions can take prerequisite arguments in an arbitrary order. In this paper, we only consider conditional witness functions that derive specifications of arguments in order from left to right.

Algorithm 1 The DUET Algorithm

Require: A SyGuS-PBE problem $(\varphi, G = \langle N, \Sigma, S, \delta \rangle)$
Require: Initial component size n
Ensure: A program $P \in L(G)$ such that $P \models \varphi$

```
1: C := {}                                          ▷ Component library C : N → 2^{L(G)}
2: repeat
3:     C := BottomUpEnum(G, C, n)                   ▷ Bottom-up enumeration for component generation
4:     S̃ := Learn(S, C, φ)                          ▷ Top-down propagation for composition
5:     if ∃P ∈ S̃. P ⊨ φ then
6:         return Pick(S̃)                           ▷ Return a solution
7:     end if
8:     n := n + 1                                   ▷ Increase the component size
9: until true
```

provided by the join nodes, which encode a cross-product of their subexpression sets. The other is provided by the DAG structure of a VSA, which allows subexpression sharing among program sets that reference the same VSA node through different paths in the DAG.

4 SYNTHESIS ALGORITHM

In this section, we describe our algorithm combining the top-down propagation and bottom-up enumerative search. We first give a high-level overview of the synthesis algorithm, describe witness functions for SyGuS language constructs, and present optimization techniques.

Notations. In the rest of this section, not only user-provided semantic specifications of SyGuS-PBE instances but also argument specifications deduced by witness functions will be represented as vectors of input-output examples. A specification φ as a vector of input-output examples $\langle i_1 \mapsto o_1, \cdots, i_n \mapsto o_n \rangle$ is often denoted by $\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}$ for conciseness where $\vec{\mathbf{e}}^{in} = \langle i_1, \cdots, i_n \rangle$ and $\vec{\mathbf{e}}^{out} = \langle o_1, \cdots, o_n \rangle$. φ^{in} (resp. φ^{out}) denotes the vector of input (resp. output) examples of φ. We will use $v[j]$ to denote the j-th element of a vector v.

4.1 Overview

Algorithm 1 shows the high-level structure of our synthesis algorithm, which takes a SyGuS-PBE problem comprising a specification φ that should be satisfied by the synthesized program as well as a regular tree grammar G as input along with an integer n meaning the maximum size of component expressions. The algorithm uses two subprocedures: the bottom-up enumerator BottomUpEnum for generating components, and the top-down composer Learn for composing the component expressions to find a solution. For each nonterminal symbol, the component library $\mathbf{C}$ stores component expressions of size $\leq n$ derivable from the nonterminal symbol. $\mathbf{C}$ is initialized to be empty (line 1), and then incrementally updated by the BottomUpEnum procedure.

The main loop (lines 2–9) is repeated until a solution is found. At the first iteration, the bottom-up enumerator BottomUpEnum generates expressions of size $\leq n$ in the grammar G. It then incrementally builds expressions of size $\leq n$ using the previously generated expressions. $\mathbf{C}$ maintains semantically unique expressions, i.e. for each nonterminal $N_i \in N$, no two expressions in $\mathbf{C}(N_i)$ are functionally equivalent with respect to the specification φ. Using the components generated so far, the top-down composer Learn constructs a VSA (line 4). If the resulting VSA contains a solution (line 5), we pick the best program from the VSA and return it as a solution (line 6). The resulting VSA may not contain a solution if the composition fails due to insufficient components.

$$\frac{}{\textsc{Learn}(N, \mathbf{C}, \varphi) = \{e \in \mathbf{C}(N) \mid e \models \varphi\}} \; \exists e \in \mathbf{C}(N).\, e \models \varphi$$

$$\frac{}{\textsc{Learn}(N, \mathbf{C}, \varphi) = \bigcup_{R \in \delta_N} \textsc{LearnRule}(N, R, \varphi)} \; \nexists e \in \mathbf{C}(N).\, e \models \varphi$$

$$\frac{\begin{array}{c}\text{ArgVSAs} = \{\langle \textsc{Learn}(N_1, \mathbf{C}, \varphi_1), \cdots, \textsc{Learn}(N_k, \mathbf{C}, \varphi_k)\rangle \mid \langle \varphi_1, \cdots, \varphi_k\rangle \in \Phi\} \\ \Phi = \{\langle \varphi_1, \cdots, \varphi_k\rangle \mid \varphi_1 \in F_1^{-1}(\varphi),\ \forall 1 < i \leq k.\ \varphi_i \in F_i^{-1}(\varphi \mid N_1 \models \varphi_1, \cdots, N_{i-1} \models \varphi_{i-1})\}\end{array}}{\textsc{LearnRule}(N, N \to F(N_1, \cdots, N_k), \mathbf{C}, \varphi) = \bigcup\{F_{\bowtie}(\widetilde{N_1}, \cdots, \widetilde{N_k}) \mid \langle \widetilde{N_1}, \cdots, \widetilde{N_k}\rangle \in \text{ArgVSAs}\}} \; F \neq \texttt{ite}$$

$$\frac{\mathbf{T} = \bigcup_{1 \leq i \leq |\varphi|} \{\textsc{Pick}(\textsc{Learn}(N_T, \mathbf{C}, \varphi[i]))\} \cup \mathbf{C}(N_T) \quad \mathbf{P} = \mathbf{C}(N_P) \quad e = \textsc{LearnDT}(\mathbf{T}, \mathbf{P}) \quad \widetilde{N} = \begin{cases} \{e\} & e \neq \bot \\ \emptyset & \text{o/w} \end{cases}}{\textsc{LearnRule}(N, N \to \texttt{ite}(N_P, N_T, N_T), \mathbf{C}, \varphi) = \widetilde{N}}$$

Fig. 4. Inference rules for constructing VSAs with the top-down propagation.

To provide more component expressions to the top-down composer, n is increased by 1 (line 8) and the process is repeated.

Our algorithm is sound and complete in that it finds a program correct with respect to the given specification φ if it exists in the search space.

Theorem 4.1. *Given a SyGuS-PBE problem $\langle \varphi, G\rangle$, Algo. 1 finds a solution if one exists in the search space $L(G)$.*

4.2 The Learn Procedure

Fig. 4 depicts constructive inference rules for the top-down deductive search. The Learn procedure takes specification φ, nonterminal N, and component library $\mathbf{C}$.

The first rule states that if there are components satisfying a given specification, the set of the component expressions is immediately returned without generating any subproblems.

The second rule states that if a component satisfying a given specification does not exist in the component library, we invoke synthesis on all production rules associated with N, and unite the resulting VSAs.

The third rule is for synthesizing conditional-free expressions. Given a rule $N \to F(N_1, \cdots, N_k)$ where F is not the if-then-else operator (`ite`), the LearnRule procedure returns a VSA where the root node is a union node. The LearnRule procedure first obtains a set of lists of VSAs (denoted ArgVSAs). Here, each list represents a set of argument expressions. To obtain ArgVSAs, we derive a set Φ of all possible combinations of argument specifications by repeatedly applying conditional witness functions. For each list of argument specifications, we obtain a list of VSAs. The resulting VSA is a union of join nodes each of which represents argument expressions.

The last rule is for synthesizing large expressions with conditionals inspired by the decision tree learning-based method of EUSolver [Alur et al. 2017]. The idea is to find different expressions that work for different subsets of the inputs, and unite them into a solution that works for all inputs. To this end, we identify *terms* (denoted $\mathbf{T}$) and *predicates* (denoted $\mathbf{P}$) separately and unify them into a single conditional expression. For example, in the `ite` expression $\texttt{ite}(x \leq y, y, x)$, the terms are x and y, and the predicate is $x \leq y$. Such a conditional expression can be represented as a decision tree. Suppose we are given a hypothesis $\texttt{ite}(N_P, N_T, N_T)$ where N_P (resp. N_T) is a nonterminal for predicates (resp. terms). The component library $\mathbf{C}$ contains expressions $\mathbf{C}(N_P)$ that can be used as predicates (resp. $\mathbf{C}(N_T)$ for terms). To construct a conditional expression, we need terms that work

Table 1. Categorization of the witness functions for the SyGuS operators. **DSF**: Domain-specific witness function generating a fresh and sufficient specification. **DSE**: Domain-specific witness function generating a pre-existing and sufficient specification. **Universal**: Domain-independent witness function generating a pre-existing specification which may not be sufficient.

	Theories			
Category	STRING	BITVEC	LIA	SAT
DSF	$\texttt{ConCat}^{-1}_{\{1,2\}}$ $\texttt{Rep}^{-1}_{\{1,2,3\}}$ $\texttt{StrToInt}^{-1}_{1}$ $\texttt{IntToStr}^{-1}_{1}$ $\texttt{SubStr}^{-1}_{\{2,3\}}$ $\texttt{StrAt}^{-1}_{2}$	$\texttt{bv}\{\texttt{add},\texttt{sub},\texttt{mul}\}^{-1}_{2}$ $\texttt{bv}\{\texttt{ashr},\texttt{lshr},\texttt{shl}\}^{-1}_{2}$	$\{+,-,\times\}^{-1}_{2}$	
DSE	$\texttt{SubStr}^{-1}_{1}$ $\texttt{StrAt}^{-1}_{1}$ $\texttt{IndexOf}^{-1}_{\{1,2\}}$	$\texttt{bvmul}^{-1}_{1}$ $\texttt{bv}\{\texttt{sdiv},\texttt{srem}\}^{-1}_{1}$ $\texttt{bv}\{\texttt{and},\texttt{or}\}^{-1}_{1}$	$\times^{-1}_{1}$ $\{/,\texttt{mod}\}^{-1}_{1}$	$\{\texttt{and},\texttt{or}\}^{-1}_{1}$
Universal	The other remaining functions			

for all inputs (i.e., each input-output example should be satisfied by at least one term expression). Since the terms supplied by the component library may not work for all inputs, we augment the set **T** of terms; we learn a term for each input-output example, and add it into **T**. And then, we attempt to learn a decision tree that correctly works on all the input-output examples. If such a decision tree exists, a singleton set containing the corresponding expression is returned. We may fail to learn a decision tree if we do not have sufficient predicates. Then, the empty set is returned.

4.3 Witness Functions for the SyGuS Language Constructs

In this section, we propose cost-effective witness functions that can be used for a wide range of SyGuS-PBE problems. Our witness functions do not always generate necessary specifications. In other words, we may fail to construct a solution even with enough component expressions available. However, this incompleteness does not harm the search completeness as already explained in Section 2.

We first introduce the following concept to classify patterns among the witness functions and categorize them.

Definition 4.2 (Fresh specification). Given a hypothesis $F(N_1, \cdots, N_k)$ about the desired program satisfying input-output examples $\varphi = \vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}$ and component library $\mathbf{C}$, a specification φ_j for j-th argument deduced by F_j^{-1} is called *fresh* if there is no component expression that can satisfy the specification. Formally, $\varphi_j (= F_j^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1, \cdots, N_{j-1} \models \varphi_{j-1}))$ is fresh iff

$$\varphi_j \cap \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_j)\} = \emptyset.$$

Otherwise, the specification is called *pre-existing*. Intuitively, if a deduced specification is pre-existing, it can be directly solved by a component expression. Otherwise, the deduced specification should be further decomposed using witness functions.

The witness functions that will be described later in this section are categorized into the following three groups as represented in Table 1.

(1) Domain-specific witness functions generating a fresh and sufficient specification:
Witness functions of this type exploit domain knowledge of underlying operators to generate a fresh and sufficient specification. For example, recall the specification $\langle i_1, i_2 \rangle \rightsquigarrow \langle$"6.775.969-238.", "82.10.56-80."$\rangle$ deduced for S_1 of $\texttt{Rep}(S_1, S_2, S_3)$ in Section 2. The specification is fresh because no component expression can satisfy it. In addition, the specification is

sufficient because specifications for the other arguments (S_2 and S_3) that lead to the entire goal (i.e., satisfying the specification for the entire expression of form $\texttt{Rep}(S_1, S_2, S_3)$) exist.

(2) Domain-specific witness functions generating a pre-existing and sufficient specification:
Witness functions of this type also exploit domain knowledge of underlying operators. For instance, recall the specification $\langle i_1, i_2\rangle \rightsquigarrow \langle$"␣+6␣775-969-238.", "␣+82␣10-56-80."$\rangle$ deduced for S of $\texttt{SubStr}(S, I_1, I_2)$ in Section 2. The specification is pre-existing because the component expression $\texttt{ConCat}(x, \text{"."})$ can satisfy it. The specification is sufficient because specifications for the other arguments (S_1 and S_2) that lead to the entire goal exist.

(3) Domain-independent witness functions generating a pre-existing specification which may not be sufficient (which we call *universal witness functions*):
Witness functions of this kind do not exploit domain knowledge of underlying operators, thereby possibly generating a specification which is not sufficient.

The universal witness functions are formally defined as follows:

Definition 4.3 (Universal witness function). Given a hypothesis $F(N_1, \cdots, N_k)$ of the desired program satisfying specification $\varphi = \vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}$ along with component library $\mathbf{C}$, the universal witness function F_j^{-1} for F that deduces a specification on j-th argument is defined as follows:

$$F_j^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1, \cdots, N_{j-1} \models \varphi_{j-1})$$
$$= \begin{cases} \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![P_j]\!]_{\vec{\mathbf{e}}^{in}} \mid P_j \in \mathbf{C}(N_j)\} & (1 \le j < k) \\ \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![P_k]\!]_{\vec{\mathbf{e}}^{in}} \mid P_k \in \mathbf{C}(N_k), \forall 1 \le i < k.\ \exists P_i \in \mathbf{C}(N_i).\ [\![F(P_1, \cdots, P_k)]\!]_{\vec{\mathbf{e}}^{in}} = \vec{\mathbf{e}}^{out}\} & (j = k) \end{cases}$$

The universal witness functions essentially enumerate all possible combinations of component expressions that can be used as arguments.

Example 4.4. Consider the following solution to a SyGuS problem[5] where the semantic specification is $\langle$"yellow", "gray", $\cdots\rangle \rightsquigarrow \langle \texttt{true}, \texttt{false}, \cdots\rangle$.

$$\texttt{Contains}(\underbrace{\texttt{ConCat}(\text{"blue"}, \texttt{ConCat}(\text{"orange"}, \text{"pink"}))}_{e_1}, \underbrace{\texttt{Rep}(x, \text{"yellow"}, \text{""})}_{e_2}) \tag{4}$$

$\texttt{Contains}(s, t)$ returns true if string s contains string t. The solution size in AST nodes is 10. To find it, we use $\texttt{Contains}_1^{-1}$ and $\texttt{Contains}_2^{-1}$, which are the universal witness functions. Suppose we are given a hypothesis $\texttt{Contains}(S_1, S_2)$ and generate a specification for S_1 using component expressions of size ≤ 5. There is no particular property of Contains that can be used to effectively restrict the value space for S_1. Thus $\texttt{Contains}_1^{-1}$ consider every component expression (including e_1 of size 5 in (4)) a potential candidate for S_1. Subsequently, for each candidate expression for S_1, $\texttt{Contains}_2^{-1}$ enumerates all the component expressions to find a correct expression for S_2. It figures out e_2 of size 4 is the one that leads to the entire goal along with e_1, thereby finding the solution.

Since enumerating expressions of size ≤ 5 is much cheaper than enumerating expressions of size ≤ 10, the solution can be found 100x faster using the universal witness functions compared to the bottom-up enumerative search for this example.

The universal witness functions may be unsound in the sense that even a specification for an argument is deduced by them, there might not exist other arguments that lead to the entire goal. For example, the grammar for the above example includes the empty string "" as a constant. The set of candidates for the first argument enumerated by $\texttt{Contains}_1^{-1}$ includes the empty string, which subsequently generates a subproblem of finding e_2 such that $\texttt{Contains}(\text{""}, e_2)$ produces the

[5] https://github.com/SyGuS-Org/benchmarks/blob/master/comp/2019/PBE_SLIA_Track/euphony/cell-equals-one-of-many-things.sl

$\texttt{ConCat}(N_1, N_2)$:

$\texttt{ConCat}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \vec{\mathbf{c}}[j] \neq \epsilon, \vec{\mathbf{c}} = \textsc{LongestCommonPrefix}(\vec{\mathbf{e}}^{out}, [\![P]\!]_{\vec{\mathbf{e}}^{in}}) \mid P \in \mathbf{C}(N_1)\}$

$\texttt{ConCat}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \varphi_1^{out}[j] \cdot \vec{\mathbf{c}}[j] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{StrToInt}(N_1)$:

$\texttt{StrToInt}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \vec{\mathbf{c}}[j] = [\![\texttt{IntToStr}(\vec{\mathbf{e}}^{out}[j])]\!]\}$

$\texttt{IntToStr}(N_1)$:

$\texttt{IntToStr}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \vec{\mathbf{c}}[j] = [\![\texttt{StrToInt}(\vec{\mathbf{e}}^{out}[j])]\!]\}$

$\texttt{SubStr}(N_1, N_2, N_3)$:

$\texttt{SubStr}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_1)\}, \forall j.\, \vec{\mathbf{e}}^{out}[j] \prec \vec{\mathbf{c}}[j]\}$

$\texttt{SubStr}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, [\![\texttt{SubStr}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j], |\vec{\mathbf{e}}^{out}[j]|)]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{SubStr}_3^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1, N_2 \models \varphi_2) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, [\![\texttt{SubStr}(\varphi_1^{out}[j], \varphi_2^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{StrAt}(N_1, N_2)$:

$$\texttt{StrAt}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \begin{cases} \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_1)\}, \forall j. \vec{\mathbf{e}}^{out}[j] \prec \vec{\mathbf{c}}[j]\} & (\forall j.\, |\vec{\mathbf{e}}^{out}[j]| = 1) \\ \emptyset & (\text{otherwise}) \end{cases}$$

$\texttt{StrAt}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \vec{\mathbf{e}}^{out}[j] = [\![\texttt{StrAt}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!]\}$

$\texttt{IndexOf}(N_1, N_2, N_3)$:

$\texttt{IndexOf}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_1)\}, \forall j.\, |\vec{\mathbf{c}}[j]| > \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{IndexOf}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_2)\}, \forall j.\, \vec{\mathbf{e}}^{out}[j] = -1 \vee$
$\vec{\mathbf{c}}[j] \preceq [\![\texttt{SubStr}(\varphi_1^{out}[j], \vec{\mathbf{e}}^{out}[j], |\varphi_1^{out}[j]| - \vec{\mathbf{e}}^{out}[j])]\!]\}$

$\texttt{IndexOf}_3^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1, N_2 \models \varphi_2) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathbf{C}(N_3)\},$
$\forall j.\, [\![\texttt{IndexOf}(\varphi_1^{out}[j], \varphi_2^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{Rep}(N_1, N_2, N_3)$:

$\texttt{Rep}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \vec{\mathbf{c}}[j] \neq \epsilon, \vec{\mathbf{c}} \in \{\textsc{GetSrc}([\![P]\!]_{\vec{\mathbf{e}}^{in}}, \vec{\mathbf{e}}^{out}) \mid P \in \mathbf{C}(N_1)\}\}$

$\texttt{Rep}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \langle(\vec{\mathbf{c}}[j], \_)\rangle = \textsc{GetRepPairs}(\varphi_1^{out}[j], \vec{\mathbf{e}}^{out}[j])\}$

$\texttt{Rep}_3^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1, N_2 \models \varphi_2) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\, \langle(\varphi_2^{out}[j], \vec{\mathbf{c}}[j])\rangle = \textsc{GetRepPairs}(\varphi_1^{out}[j], \vec{\mathbf{e}}^{out}[j])\}$

Others $F(N_1, \cdots, N_k)$: (Universal witness function)

Fig. 5. Witness functions for strings. $\mathbf{C}$ denotes a given component library, and $\cdot$ denotes the string concatenation operator.

desired output examples $\langle \texttt{true}, \texttt{false}, \cdots \rangle$. Because the empty string cannot contain any strings, there is no such e_2. On the other hand, domain-specific witness functions are sound by exploiting domain-knowledge of underlying operators. The following theorem states that the domain-specific witness functions listed in Table 1 are sound.

Theorem 4.5. *Every specification deduced by a domain-specific witness function in Table 1 is sufficient.*

In the sequel, we describe our witness functions for the operators supported in theories of strings, fixed-width bitvectors, linear integer arithmetic, and propositional logic in detail.

STRING. Fig. 5 depicts the definitions of our witness functions for strings. All the auxiliary functions used in the definitions are defined in Fig. 6. We illustrate how some of them work on the following SyGuS-PBE problem.

```
Function LongestCommonPrefix(s⃗, t⃗)
return c⃗ such that ∀j. c⃗[j] is the longest common prefix of s⃗[j] and t⃗[j].
Function replaceWithRepPairs(s, p)
u := s
for i = 1 to |p| do
    (s_match, s_replace) := p[i]
    u := ⟦Rep(u, s_match, s_replace)⟧
end for
return u
Function getSrc(s⃗, t⃗)
c⃗ := ⟨⟩
for i = 1 to |s⃗| do
    p := getRepPairs(s⃗[i], t⃗[i])
    if p is empty then
        c⃗ := c⃗ + ⟨ϵ⟩
    else
        p′ := p[1, ···, |p| − 1]
        c⃗ := c⃗ + ⟨replaceWithRepPairs(s⃗[i], p′)⟩
    end if
end for
return c⃗
Function getRepPairs(s, t)
p := ⟨⟩                          ▷ Vector of replacement pairs
s°, t° := Align(s, t)            ▷ Get an alignment of s and t
if EditDistance(s°, t°) ≥ min(|s|, |t|) then
    return ⟨⟩                    ▷ s and t are not similar enough
end if
i := 1    s_sub := ϵ    t_sub := ϵ
for i = 1 to |s°| do
    if s°[i] = t°[i] then
        delete all ϵ from s_sub, t_sub.
        p := p + ⟨(s_sub, t_sub)⟩   ▷ +: vector concatenation
        s_sub := ϵ    t_sub := ϵ
    else
        s_sub := s_sub · s°[i]      ▷ ·: string concatenation
        t_sub := t_sub · t°[i]
    end if
end for
if t = replaceWithRepPairs(s, p) then
    return p
else
    return ⟨⟩
end if
```

Fig. 6. Auxiliary functions used in Fig. 5.

Example 4.6. Consider the following problem of synthesizing a string-manipulating function f that takes as input a string x. The syntactic specification for f is the grammar:

$$\begin{aligned} S &\rightarrow x \mid \text{“}\_\text{”} \mid \text{“Dr.”} \mid \texttt{SubStr}(S, I, I) \mid \texttt{ConCat}(S, S) \\ I &\rightarrow 0 \mid \texttt{IndexOf}(S, S, I) \end{aligned}$$

where $\texttt{IndexOf}(s, t, i)$ returns the position of the first occurrence of t in the string s after the index i. It returns -1 if t does not occur in s. The semantic specification for f is $\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}$ where $\vec{\mathbf{e}}^{in} = \langle$“Jan␣K”, “Nancy␣F”$\rangle$ and $\vec{\mathbf{e}}^{out} = \langle$“Dr.␣Jan”, “Dr.␣Nancy”$\rangle$. The solution is

$$\texttt{ConCat}(\text{“Dr.”}, \texttt{ConCat}(\text{“}\_\text{”}, \texttt{SubStr}(x, 0, \texttt{IndexOf}(x, \text{“}\_\text{”}, 0)))).$$

We first generate component expressions of size 1 and obtain the following component library:

$$\mathbf{C} = \{S \mapsto \{x, \text{“}\_\text{”}, \text{“Dr.”}\}, I \mapsto \{0\}\}.$$

And then, suppose we first consider a hypothesis $\texttt{ConCat}(S_1, S_2)$. We obtain specifications for arguments S_1 and S_2 as follows (LCP denotes LongestCommonPrefix for brevity):

$$\begin{aligned} &\texttt{ConCat}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \text{LCP}(\langle\text{“Dr.␣Jan”, “Dr.␣Nancy”}\rangle, \langle\text{“Dr.”, “Dr.”}\rangle)\} \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Dr.”, “Dr.”}\rangle\} \end{aligned} \qquad \begin{aligned} &\texttt{ConCat}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid S_1 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Dr.”, “Dr.”}\rangle) \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣Jan”, “␣Nancy”}\rangle\} \end{aligned}$$

The specification for S_1 is directly satisfied by the component “Dr.”. However, the specification for S_2 is not solvable with any of the components. We further decompose it. Suppose again we consider the hypothesis $\texttt{ConCat}(S_1, S_2)$. We obtain specifications for arguments S_1 and S_2 as follows:

$$\begin{aligned} &\texttt{ConCat}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣Jan”, “␣Nancy”}\rangle) \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \text{LCP}(\langle\text{“␣Jan”, “␣F”}\rangle, \langle\text{“␣”, “␣”}\rangle)\} \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣”, “␣”}\rangle\} \end{aligned} \qquad \begin{aligned} &\texttt{ConCat}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣Jan”, “␣Nancy”}\rangle \mid S_1 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣”, “␣”}\rangle) \\ &= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan”, “Nancy”}\rangle\} \end{aligned}$$

The specification for S_1 is satisfied by the component "␣". The specification for S_2 is further decomposed. Suppose we consider a hypothesis $\texttt{SubStr}(S, I_1, I_2)$. We obtain specifications for arguments as follows:

$$\begin{aligned}
&\texttt{SubStr}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan”, “Nancy”}\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![x]\!]_{\vec{\mathbf{e}}^{in}}\} \quad (\because x \in \mathbf{C}(S) \text{ produces strings containing “Jan” and “Nancy” respectively.})\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle\}\\
&\texttt{SubStr}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan”, “Nancy”}\rangle \mid S \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 0, 0\rangle\} \quad (\because [\![\texttt{SubStr}(\text{“Jan␣”}, \mathbf{0}, 3)]\!] = \text{“Jan”}, [\![\texttt{SubStr}(\text{“Nancy␣”}, \mathbf{0}, 5)]\!] = \text{“Nancy”}.)\\
&\texttt{SubStr}_3^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan”, “Nancy”}\rangle \mid S \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle, I_1 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle 0, 0\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 3, 5\rangle\}
\end{aligned}$$

The specifications for S and I_1 are satisfied by x and 0, respectively. The specification for I_2 is further decomposed. Suppose we consider a hypothesis $\texttt{IndexOf}(S_1, S_2, I)$. We obtain specifications for arguments as follows:

$$\begin{aligned}
&\texttt{IndexOf}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 3, 5\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![x]\!]_{\vec{\mathbf{e}}^{in}}\} \quad (\because x \in \mathbf{C}(S) \text{ produces strings of which lengths are greater than 3 and 5 resp.})\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle\}\\
&\texttt{IndexOf}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 3, 5\rangle \mid S_1 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![\text{“␣”}]\!]_{\vec{\mathbf{e}}^{in}}\} \quad (\because \text{ The position of “␣” in “Jan␣K” and “Nancy␣F” are 3 and 5 resp.})\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣”, “␣”}\rangle\}\\
&\texttt{IndexOf}(\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 3, 5\rangle \mid S_1 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“Jan␣K”, “Nancy␣F”}\rangle, S_2 \models \vec{\mathbf{e}}^{in} \rightsquigarrow \langle\text{“␣”, “␣”}\rangle)\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow [\![0]\!]_{\vec{\mathbf{e}}^{in}}\} \quad (\because 0 \in \mathbf{C}(I) \text{ produces proper starting positions.})\\
&= \{\vec{\mathbf{e}}^{in} \rightsquigarrow \langle 0, 0\rangle\}
\end{aligned}$$

The specifications for S_1, S_2, and I are satisfied by component expressions x, "␣", and 0, respectively. Because no subproblem is left unsolved, the solution is found.

Next, we detail the witness functions for the Rep operator which are based on the following concept.

Definition 4.7 (Alignment [Durbin et al. 1998]). Suppose we are given a fixed set of alphabets Σ such that $\epsilon \notin \Sigma$. A pair of words $a^\diamond, b^\diamond \in (\Sigma \cup \{\epsilon\})^+$ is called alignment of sequences a and b ($a^\diamond$ and $b^\diamond$ are called alignment strings) iff

- $|a^\diamond| = |b^\diamond|$
- $\forall 1 \leq i \leq |a^\diamond|.\ a_i^\diamond \neq \epsilon \vee b_i^\diamond \neq \epsilon$
- Deleting all ϵ from $a^\diamond$ (resp. $b^\diamond$) yields a (resp. b).

The getRepPairs procedure in Fig. 6 computes alignments and replacement pairs (introduced in Section 2) of given two strings. A complication is that there may be infinitely many alignments of aribtrary two strings. We use Needleman-Wunsch algorithm [Needleman and Wunsch 1970] to find the best alignment which incurs the smallest edit distance.

The procedure does not return replacement pairs if two given strings are not similar enough (line 24). This is a heuristic for enforcing termination of the TDP by avoiding deducing infinitely many specifications due to repeated string replacements. The procedure checks correctness of generated replacement pairs (line 38), which is necessary in the following case.

Example 4.8. Let s = "+6.775-969-238" and t = "+6-775␣969-23". The best alignment of s and t is "+6.775-969-238" and "+6-775␣969-23ϵ". The replacement pairs is $\langle$(".", "-"), ("-", "␣"), ("8", ϵ)$\rangle$.

String replacement on s using the replacement pairs yields "+6␣775-969-23", which is not equivalent to t. In such a case, the GetRepPairs procedure simply returns the empty list, and no argument specifications are deduced.

$\texttt{bvadd}(N_1, N_2)$:
$\texttt{bvadd}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}\}$
$\texttt{bvadd}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ \vec{\mathbf{c}}[j] = [\![\texttt{bvsub}(\vec{\mathbf{e}}^{out}[j], \varphi_1^{out}[j])]\!],\ |\vec{\mathbf{c}}[j]| < |\vec{\mathbf{e}}^{out}[j]|\}$

$\texttt{bvsub}(N_1, N_2)$:
$\texttt{bvsub}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}\}$
$\texttt{bvsub}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ \vec{\mathbf{c}}[j] = [\![\texttt{bvsub}(\varphi_1^{out}[j], \vec{\mathbf{e}}^{out}[j])]\!],\ |\vec{\mathbf{c}}[j]| < |\vec{\mathbf{e}}^{out}[j]|\}$

$\texttt{bvmul}(N_1, N_2)$:
$\texttt{bvmul}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \forall j.\ |\vec{\mathbf{c}}[j]| \neq 1, [\![\texttt{bvsrem}(\vec{\mathbf{e}}^{out}[j], \vec{\mathbf{c}}[j])]\!] = 0\}$
$\texttt{bvmul}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ [\![\texttt{bvsdiv}(\vec{\mathbf{e}}^{out}[j], \varphi_1^{out}[j])]\!] = \vec{\mathbf{c}}[j]\}$

$\texttt{bvsdiv}(N_1, N_2)$:
$\texttt{bvsdiv}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \forall j.\ |\vec{\mathbf{c}}[j]| \geq |\vec{\mathbf{e}}^{out}[j]|\}$
$\texttt{bvsdiv}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_2)\}, \forall j.\ [\![\texttt{bvsdiv}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvsrem}(N_1, N_2)$:
$\texttt{bvsrem}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \forall j.\ |\vec{\mathbf{c}}[j]| \geq |\vec{\mathbf{e}}^{out}[j]|\}$
$\texttt{bvsrem}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_2)\}, \forall j.\ [\![\texttt{bvsrem}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvand}(N_1, N_2)$:
$\texttt{bvand}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \forall j, i.\ [\vec{\mathbf{e}}^{out}[j]]_i = 1 \implies [\vec{\mathbf{c}}[j]]_i = 1\}$
$\texttt{bvand}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_2)\}, \forall j.\ [\![\texttt{bvand}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvor}(N_1, N_2)$:
$\texttt{bvor}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \forall j, i.\ [\vec{\mathbf{e}}^{out}[j]]_i = 0 \implies [\vec{\mathbf{c}}[j]]_i = 0\}$
$\texttt{bvor}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_2)\}, \forall j.\ [\![\texttt{bvor}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvashr}(N_1, N_2)$:
$\texttt{bvashr}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \exists 1 \leq i < w.\ [\![\texttt{bvashr}(\vec{\mathbf{c}}[j], i)]\!] = \vec{\mathbf{e}}^{out}[j]\}$
$\texttt{bvashr}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ [\![\texttt{bvashr}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvlshr}(N_1, N_2)$:
$\texttt{bvlshr}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \exists 1 \leq i < w.\ [\![\texttt{bvlshr}(\vec{\mathbf{c}}[j], i)]\!] = \vec{\mathbf{e}}^{out}[j]\}$
$\texttt{bvlshr}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ [\![\texttt{bvlshr}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

$\texttt{bvshl}(N_1, N_2)$:
$\texttt{bvshl}_1^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out}) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \vec{\mathbf{c}} \in \{[\![P]\!]_{\vec{\mathbf{e}}^{in}} \mid P \in \mathrm{C}(N_1)\}, \exists 1 \leq i < w.\ [\![\texttt{bvshl}(\vec{\mathbf{c}}[j], i)]\!] = \vec{\mathbf{e}}^{out}[j]\}$
$\texttt{bvshl}_2^{-1}(\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{e}}^{out} \mid N_1 \models \varphi_1) = \{\vec{\mathbf{e}}^{in} \rightsquigarrow \vec{\mathbf{c}} \mid \forall j.\ [\![\texttt{bvshl}(\varphi_1^{out}[j], \vec{\mathbf{c}}[j])]\!] = \vec{\mathbf{e}}^{out}[j]\}$

Others $F(N_1, \cdots, N_k)$: (Universal witness function)

Fig. 7. Witness functions for bitvectors of fixed bit-width w.

BITVEC. Fig. 7 depicts the definitions of our witness functions for the theory of bitvectors of fixed-width w. We follow the standard syntax and semantics of the bit-vector operators described in the SMT-LIB v2.0 standard [Barrett et al. 2010]. We denote $[n]_i$ as i-th bit of the bitstring representation of n (i.e., $[n]_i \stackrel{\text{def}}{=} (n/2^i) \mod 2$).

LIA and Propositional Logic. Our witness functions for LIA and SAT are defined similarly to the witness functions for bitvectors. For example, our witness functions for integer and boolean

operations are defined simiarly to the corresponding ones for bitvectors (e.g., $+^{-1}_{\{1,2\}}$ are similar to $\mathsf{bvadd}^{-1}_{\{1,2\}}$).

We use a heuristic to enforce termination of the TDP for deducing numeric output examples. The witness functions $+^{-1}_2$ and $-^{-1}_2$ (therefore bvadd^{-1}_2 and bvsub^{-1}_2 as well) deduce output examples for arguments only when their absolute values are smaller than those of given original output examples.

Example 4.9. Recall the given hypothesis $I_1 - I_2$ when a desired output is 14 in Section 2. Suppose we set the expected output of I_1 to be 1, then the expected output of I_2 is determined to be -13. To solve the subproblem for I_2, suppose we again use the hypothesis $I_1 - I_2$. If we set the expected output of I_1 to be 1 again, then the expected output of I_2 is determined to be 14, which is the initial problem. In this manner, we may repeatedly revisit the same subproblem unless we use the heuristic to enforce termination. Using the heuristic, we do not revisit the initial problem since $|14|$ is greater than $|-13| = 13$.

The Learn procedure using the witness functions described so far is guaranteed to terminate.

Theorem 4.10. *Given a SyGuS-PBE instance where the background theory is a combination of theories of string, fixed-width bitvectors, LIA, and SAT, the Learn procedure eventually terminates.*

4.4 Optimizations

We perform useful optimizations when running the Learn procedure. We cache synthesis results for every distinct learning subproblem, which makes our algorithm an instance of dynamic programming by avoiding solving the same subproblems multiple times. We also maintain an additional cache to record subproblems that we are trying to solve (but not solved yet) which is useful in the following case. Given a specification, suppose the witness function $\mathtt{StrToInt}^{-1}_1$ deduces an argument specification, which is fed into $\mathtt{IntToStr}^{-1}_1$ as input subsequently. The functon $\mathtt{IntToStr}^{-1}_1$ may generate the original specification that $\mathtt{StrToInt}^{-1}_1$ initially took. To break such a cycle, whenever the Learn procedure is given a problem which exists in the additional cache, it does not solve it by returning the empty VSA. Lastly, we bound the maximum height of VSAs. Though our witness functions always construct VSAs of finite height, the height may be huge if numeric values that appear in user-provided examples are large. For the experiment, we set the maximum height of VSAs to be 15, which is enough to avoid generating colossal VSAs.

5 EVALUATION

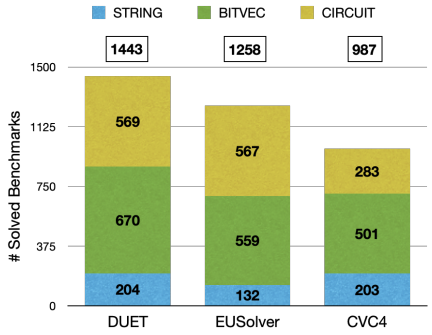
We have implemented our approach in a tool called Duet[6] which consists of 5K lines of OCaml code and employs *Z*3 [De Moura and Bjorner 2008] as the constraint solving engine. Our tool is available for download at https://github.com/wslee/duet.

This section evaluates our Duet system to answer the questions:

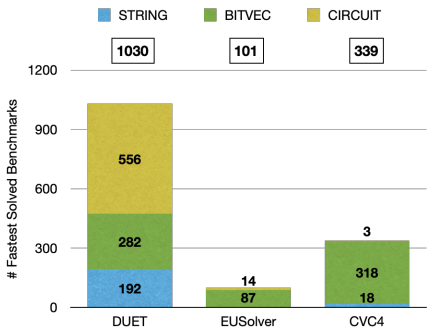
- **Q1:** How does Duet perform on synthesis tasks from a variety of different application domains?
- **Q2:** How does Duet compare with existing general-purpose and domain-specific synthesis techniques?
- **Q3:** What are the sizes of necessary components to construct solutions?

All of our experiments were conducted on Linux machines with Intel Xeon 2.6GHz CPUs and 256G of memory.

[6] **D**omain-**U**naware inductive synthesis by combining **E**numeration and **T**op-down propagation



(a) # Solved benchmarks.



(b) # Fastest solved benchmarks.

	# Solved			Time (Average)			Time (Median)			Size (Average)			Size (Median)		
Domain	**Duet**	**CVC**	**EU**	**D**	**C**	**E**	**D**	**C**	**E**	**D**	**C**	**E**	**D**	**C**	**E**
String	204	203	132	4.1	43.1	344.2	<0.1	0.2	0.9	27.6	225.4	7.3	12.5	21	7
Bitvec	670	501	559	39.2	197.6	317.5	6.7	2.1	11.9	361.2	476.6	326.6	53.5	92	52
Circuit	569	283	567	21.5	128.7	327.1	0.3	5.1	33.6	10.5	9.4	9.6	11	11	11
Overall	**1443**	**987**	**1258**	**27.3**	**146.0**	**324.6**	**4.0**	**2.2**	**11.5**	**175.7**	**290.9**	**150.2**	**15**	**26**	**12**

(c) Statistics for the solving times and solution sizes. All times are in seconds. The number of benchmark problems for the String, Bitvec and Circuit domains are **205**, **750**, **581**, respectively (**1536** in total).

Fig. 8. Main result comparing the performance of Duet, EUSolver and CVC4 (breakdown by domains). The timeout is set to one hour.

5.1 Experimental Setup

Benchmarks. We use all benchmarks used for evaluating the statistical model-guided synthesizer Euphony [Lee et al. 2018] [7] and the program synthesis-based circuit optimizer for homomorphic evaluation Lobster [Lee et al. 2020]. The collected synthesis tasks are from three application domains: i) string manipulation (String), ii) bit-vector manipulation (Bitvec), and iii) circuit transformation (Circuit).

The **String** benchmarks contain 205 tasks, including 108 from the SyGuS competition, 37 queries by spreadsheet users in StackOverflow, and 60 articles about Excel programming in Exceljet. The background theory is SLIA (String + LIA). All the benchmarks correspond to common data manipulation tasks in spreadsheet and input-output examples are provided for each benchmark. The semantic specifications comprise 2 – 400 examples.

The **Bitvec** benchmarks comprise 750 problems from the SyGuS competition. The background theory is BV. These problems aim at finding programs equivalent to randomly generated bit-manipulating programs from input-output examples. The benchmarks are motivated by program deobfuscation [Jha et al. 2010]. The semantic specifications comprise 10 – 1000 examples.

The **Circuit** benchmarks comprise 581 problems. The background theory is SAT. Among them, 212 problems used in the SyGuS competition are motivated by side-channel attacks on cryptographic modules in embedded systems [Eldib et al. 2016]. Each problem is, given a circuit C, to synthesize a constant-time circuit C' (i.e. resilient to timing attacks) that behaves the same as C. The other 369 problems are from [Lee et al. 2020]. These problems are motivated by optimizing homomorphic evaluation circuits. Each problem is, given a circuit C, to synthesize a circuit C' of smaller *multiplicative depth* (the maximal number of consecutive AND operations) that is

[7] The benchmarks are a "harder version" of the SyGuS competition benchmarks. The syntactic restriction in each problem is replaced by a more general grammar. For example, the grammar for the SyGuS'18 competition (PBE-BITVEC track) comprises 13 production rules, whereas the grammar we used comprises 38 rules.

functionally equivalent to C. The semantic specification is a boolean formula expressing the functional equivalence.

Baseline Solvers. We compare DUET to existing synthesis tools. For all of the three domains, we compare with general-purpose tools EUSOLVER and CVC4. EUSOLVER won the 2016 SyGuS competition, and CVC4 won the 2017 – 2019 competitions [Past SyGuS Competition 2020]. Especially, the comparison with EUSOLVER is to confirm benefits of combining the TDP and bottom-up enumeration. To confirm how DUET compares against domain specialization techniques, we also compare against EUPHONY [Lee et al. 2018], a probabilistic model-guided synthesis tool for all of the three domains. EUPHONY itself is a general-purpose tool, but it can be specialized for specific domains by using learned probabilistic models and custom feature maps. Probabilistic models can themselves be viewed as a kind of domain specialization, and furthermore, the feature maps designed for the three domains allow such models to generalize well across synthesis problems within a domain when their solutions have different probability distributions.

5.2 Effectiveness of DUET

We evaluate DUET on synthesis problems from all three domains and compare it with EUSOLVER and CVC4. The initial component size n for DUET is set to be 1 for the STRING and CIRCUIT domains and 3 for the BITVEC domain. For each instance, we measure the running time of DUET and the size of the synthesized program, using a timeout of one hour.

The results are summarized in Fig. 8 where (8a) – (8b) show the numbers of benchmark problems solved, and the solved with the fastest solving time respectively, and (8c) shows detailed statistics for the synthesis times and solution sizes. DUET outperforms the other baselines in terms of the number of solved problems: DUET is able to solve 1443 out of 1536 problems from the three domains cumulatively, whereas EUSOLVER and CVC4 are able to solve 1258 and 987 problems, respectively. Furthermore, DUET significantly outperforms the other baselines in terms of synthesis time: DUET is the fastest solver in 1030 problems, whereas EUSOLVER and CVC4 are the fastest only in 101 and 339 problems, respectively.

We also measure solution quality by solution sizes in AST nodes. The solution size is not necessarily proportional to the difficulty. For instance, given a problem comprising m input-output examples as a semantic specification and a syntactic specification that allows to use conditionals, a useless solution is a conditional expression with m conditionals. Therefore, smaller solutions are better in that they are not results of overfitting to the given input-output examples. EUSOLVER generates the smallest solutions in general (the average and median sizes are 150 and 12), but DUET also generate solutions of similar sizes: the average and median sizes are 176 and 15. On the other hand, CVC4 often generate large solutions: the average and median sizes are 291 and 26.

Next, we study the results for each domain in detail. Table 2 shows the detailed results on randomly chosen 30 problems (10 for each domain) solved by DUET, uniformly distributed over solution sizes.

Result for STRING. DUET outperforms the other baseline solvers in terms of both scalability and solution quality. Out of 205 problems, DUET could solve 204 problems, with average and median times of 4.1s and 0.02s. We investigate the only one unsolved problem (`univ_6-long-repeat.sl`) and conclude the synthesis problem is unrealizable (i.e., no solution exists). Therefore, DUET virtually solved all the problems. DUET finds 98% of these solutions within a minute. On the other hand, EUSOLVER could solve 132 problems, with average and median times of 6m 44s and 0.9s.

Table 2. Results for 30 randomly chosen benchmark problems (10 for each domain), where $|E|$ shows the number of examples, **Time** gives synthesis time, $\mathbf{T}_{\text{TD}}$ (resp. $\mathbf{T}_{\text{BU}}$) gives time spent for Top-down propagation (resp. Bottom-up enumeration), $|P|$ shows the size of the synthesized program (measured by number of AST nodes), n gives the maximum size of component expressions used to construct a solution, and $|C|$ denotes the number of expressions in the component library.

			EUSolver		CVC4		Duet					
Domain	**Benchmark**	$\|E\|$	**Time**	$\|P\|$	**Time**	$\|P\|$	**Time**	$\mathbf{T}_{\text{TD}}$	$\mathbf{T}_{\text{BU}}$	$\|P\|$	n	$\|C\|$
String	stackoverflow1	3	1887.6	9	33.8	44	0.05	0.03	0.01	11	3	118
	stackoverflow2	2	2789.6	12	1.3	31	0.02	0.01	0.01	15	2	21
	stackoverflow3	3	1291.3	10	234.2	122	0.02	0.01	0.01	16	2	27
	stackoverflow4	3	> 1h	–	10.6	162	0.05	0.02	0.03	13	4	238
	exceljet1	3	971.8	10	89.1	106	0.03	0.01	0.02	62	2	23
	exceljet2	3	> 1h	–	17.2	48	0.03	0.01	0.02	76	2	20
	exceljet3	4	> 1h	–	33.2	222	0.03	0.02	0.01	23	3	46
	exceljet4	4	> 1h	–	53.1	522	0.05	0.03	0.02	25	3	62
	phone-10-long-repeat	400	> 1h	–	109.4	135	0.10	0.03	0.07	14	4	422
	phone-9-long-repeat	400	> 1h	–	14.9	7171	0.03	0.01	0.02	18	1	13
Bitvec	113_10	10	3.0	25	0.5	25	4.4	4.2	0.2	15	3	464
	116_1000	1000	1464.0	2113	> 1h	–	65.5	54.9	9.9	3871	3	464
	icfp_gen_10.1	31	16.0	40	1.4	19	7.8	7.7	0.1	34	3	485
	icfp_gen_12.18	42	> 1h	–	> 1h	–	5.3	5.2	0.1	113	3	469
	10_1000	1000	985.8	2215	491.6	2218	5.8	5.5	0.3	19	3	464
	100_1000	1000	754.2	1884	563.7	2867	67.7	57.4	10.3	691	3	464
	146_1000	1000	1444.1	2141	80.4	3027	72.9	62.5	10.4	1937	3	464
	icfp_gen_15.13	25	> 1h	–	> 1h	–	34.8	33.9	0.9	42	4	1890
	12_10	10	235.5	22	868.3	16	30.8	30	0.8	46	4	1786
	65_10	10	> 1h	–	139.0	75	> 1h	–	–	–	5	44023
Circuit	CrCy10-sbox2-D5-102	–	174.8	13	3.6	13	0.2	0.01	0.03	13	6	240
	CrCy10-sbox2-D5-104	–	575.9	15	52.2	16	5.2	0.3	4.1	15	9	3405
	CrCy10-sbox2-D5-14	–	360.2	14	15.4	15	0.3	0.01	0.11	15	7	449
	CrCy8-P12-D9-1	–	1690.7	17	14.8	17	10.9	0.04	8.95	21	11	4717
	CrCy8-P12-D5-3	–	> 1h	–	21.1	17	1.5	0.01	1.11	21	10	2216
	avg-opt_59_4	–	799.7	11	43.2	11	1.4	0.06	1.1	11	5	1217
	avg-opt_50_2	–	2413.3	12	> 1h	–	300.4	1.5	288.7	12	8	31086
	hd02.eqn_19_2	–	1050.1	11	2469.5	13	10.4	1.24	8.53	11	5	2282
	sorting_naive_319_2	–	2455.3	12	> 1h	–	25.9	1	24.14	13	6	6608
	insertion-3in_24_1	–	1424.1	14	> 1h	–	57.7	0.3	55.3	17	8	8968

Duet outperforms EUSolver on all the problems. CVC4 could solve 203 problems, with average and median times of 43s and 0.2s.

Ostensibly Duet and CVC4 show similar performance in terms of synthesis time, however, the major difference is in the quality of solutions. On the contrary to Duet and EUSolver which prioritize small solutions, CVC4 sometimes generates large and unreadable solutions which are "overfitted" to provided input-output examples. For example, for the problem `phone-9-long-repeat`, CVC4 generates a solution of size 7171 whereas Duet generates a solution of size only 18. The average and median sizes of solutions found by Duet are 28 and 13, whereas those of CVC4 are 225 and 21. Both the average and median sizes of solutions found by EUSolver are smaller (7), but it is due to the limited scalability.

Result for Bitvec. Also in the Bitvec domain, Duet outperforms the other baseline solvers in terms of scalability. Out of 750 problems, Duet could solve 670 problems, with average and median times of 39s and 7s. EUSolver could solve 559 problems with average and median times of 5m 18s and 12s, and CVC4 could solve 501 problems with average and median times of 3m 19s and 2s.

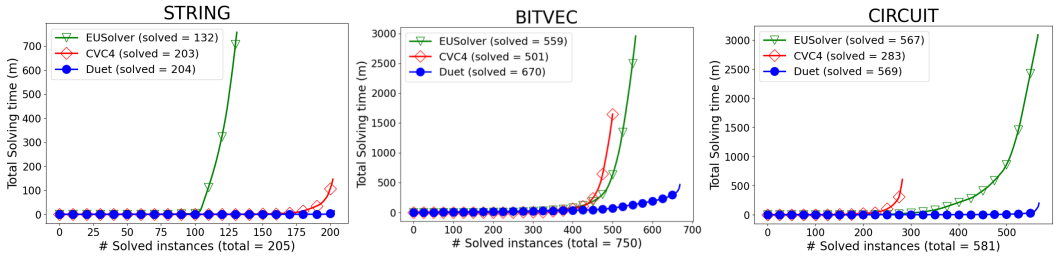


Fig. 9. Comparison between DUET and the other baseline solvers (CVC4 and EUSOLVER) on different domains.

Table 3. Result comparing the performance of DUET and EUPHONY.

	# Problems			# Solved		Time (Average)		Time (Median)		Size (Average)		Size (Median)	
Domain	Total	Train	Test	DUET	EUPH	DUET	EUPH	DUET	EUPH	DUET	EUPH	DUET	EUPH
STRING	205	123	82	81	27	10.3	211.5	0.04	1	50.5	12.2	29	11
BITVEC	750	461	289	209	243	89.2	719.6	54.6	137.9	966.6	860.2	164	116.3
CIRCUIT	581	485	96	84	54	88.6	2049.3	25.6	3226.9	14.2	23.9	13	30
Overall	1536	1069	467	374	324	71.9	898.8	25.3	191.8	554.3	650.1	45	66

The average and median sizes of solutions found by DUET are 361 and 54. Those of CVC4 are 477 and 92. Those of EUSOLVER are 327 and 52. Thus, DUET generates solutions comparably as small as solutions generated by EUSOLVER.

Result for CIRCUIT. DUET is again the best in terms of scalability. In terms of solution quality, all three solvers are similar. Out of 581 problems, DUET could solve 569 problems, with average and median times of 22s and 0.5s. EUSOLVER could solve 567 problems, with average and median times of 5m 27s and 34s. CVC4 could solve 283 problems, with average and median times of 2m 9s and 5s. The average and median sizes of solutions found by all the tools are 10 and 11, respectively.

Analysis of Overfitting. We manually inspect solutions of the 30 benchmark problems in Table 2 found by DUET to investigate if DUET is prone to overfitting. In the STRING domain, 7 out of 10 (except `stackoverflow3.sl` and `exceljet{2,4}.sl`) solutions are the desired programs. In the BITVEC domain, we suspect 4 out of 10 are the desired ones ("suspect" because the correct solutions are unknown). On the other hand, overfitting does not occur in the CIRCUIT domain because the semantic specifications are logical specifications.

We can mitigate overfitting by letting DUET find *all* solution candidates that can be found with a current set of component expressions and choose the smallest one among them (instead of stopping the search when a solution is found). Furthermore, providing a richer set of initial component expressions by setting n to be a larger number also helps. That is because the more component expressions are available, the broader space of programs is explored by DUET. For example, DUET initially found a non-desired solution of size 25 for `exceljet4.sl`. When we provide a richer set of initial components by setting n to be 4 and let DUET pick the smallest one among all consistent programs, DUET can find the desired program of size 11 at the cost of extra overhead (0.5 sec).

Summary of Results. DUET solves harder synthesis problems more quickly compared to state-of-the-art baseline tools in diverse domains. The three evaluated domains not only exercise different SMT theories but also different kinds of specifications (PBE vs. logical) of the desired programs.

5.3 Comparison to Euphony

We compare Duet with Euphony which learns statistical models from easily obtainable solutions. For each domain, we use all problems that EUSolver could solve within 10 minutes each as the training set, and we train the model for that domain using the solutions found by EUSolver as done in [Lee et al. 2018]. The training set comprises 1069 (~ 70%) of our benchmark problems.

The result is summarized in Table 3. In terms of the overall number of solved problems, Duet is better than Euphony. Duet significantly outperforms Euphony in the domains of String and Circuit. In those domains, Euphony suffers from limited scalability when attempting to synthesize large and conditional-free expressions, whereas Duet can successfully scale to such expressions through the divide-and-conquer. However, in the Bitvec domain, Euphony captures the statistical regularity that exists in the solutions and effectively guides the search, thereby outperforming Duet. Also, Euphony avoids overfitting in PBE settings by virtue of guiding synthesis toward more likely programs, thereby generating smaller solutions than Duet.

Next, we provide Euphony with a more favorable setting where a richer training set is given. For each domain, we generate 5 random samples for training sets each of which comprises 75% of the benchmarks, and use a learned model which is the best in terms of the number of testing (the other remaining) instances solved by Euphony. In the String domain, Euphony solved 44 out of 51 whereas Duet solved 50. In the Bitvec domain, Euphony solved 149 out of 187 whereas Duet solved 167. In the Circuit domain, Euphony solved 112 out of 145 whereas Duet solved 131. Thus, Duet still outperforms Euphony.

Overall, our results show that our approach provides significant performance gains that are complementary to those achieved by Euphony, and it is promising to incorporate our approach into such domain specializations.

5.4 Analysis of Component Sizes

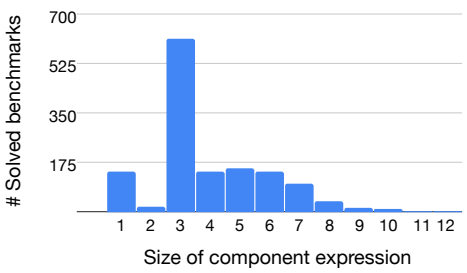


Fig. 10. # Solved benchmark per component size.

Figure 10 shows the frequency distribution of benchmarks according to their maximum sizes of component expressions when Duet finds a solution. Overall, the maximum size of components range from 1 to 12 with an average of 4. For the String domain, the maximum size ranges from 1 to 6 with an average of 1.8. For the Bitvec domain, the maximum size ranges from 3 to 5 with an average of 3.1. For the Circuit domain, the maximum size ranges from 3 to 12 with an average of 5.9. Out of solved 1443 problems in all the domains, 92% of problems (1317) could be solved with component expressions of size ≤ 7. Such small expressions can be quickly explored by the bottom-up enumeration in practice. This indicates that Duet could compose large solutions using small expressions.

6 RELATED WORK

In this section, we discuss related work on program synthesis techniques about combining deduction and enumeration, bidirectional enumerative search strategies and decomposing synthesis problems.

Combining Deduction and Enumeration. Various previous methods have been proposed to use deduction to effectively prune the search space explored by enumeration [Feng et al. 2018, 2017;

Feser et al. 2015; Frankle et al. 2016; Lee et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016; So and Oh 2017; Wang et al. 2017]. While enumerating solution candidates, these approaches use a deduction engine to determine whether or not a currently considered candidate (mostly in the form of a *partial program* with holes) is feasible (i.e., there may exist a solution derivable from the partial program). Powerful deduction engines such as type checkers [Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016], constraint solvers [Feng et al. 2018, 2017], or abstract interpreters [So and Oh 2017; Wang et al. 2017] have been used for pruning the search space. On the contrary to these methods, we use systematic enumeration to guide deduction instead of using deductive reasoning to accelerate enumeration.

Our method is also different from another line of work that uses deduction and enumeration separately without a synergistic combination. In FlashFill, candidate regular expressions are constructed by enumeration, and they are used when they can directly solve decomposed subproblems during top-down synthesis (cf. the `GeneratePosition` procedure (Fig. 7) in [Gulwani 2011]). Here, the decomposition happens irrespective of what regular expressions are available. It would be more desirable to guide the decomposition to generate subproblems that can be eventually solved by the available expressions. Our witness function for the `Rep` operator described in Section 2 demonstrates this guided decomposition. It deduces desired outputs of the first argument similar to the output of `ConCat`(x, "."), which is one of the available component expressions. FlashMeta [Polozov and Gulwani 2015] basically uses deduction to repeatedly decompose a synthesis task until all subdivided tasks are easily solvable, but it occasionally switches to enumerative search when it is likely to be more efficient than deduction. DryadSynth [Huang et al. 2020] repeatedly performs custom divide-and-conquer methods and tries to solve subproblems first by deduction, and then enumeration as the last resort if the deduction is not applicable. In these methods, enumeration and deduction are independent of each other, whereas our method is based on a synergistic combination of the top-down deductive search and bottom-up enumerative search.

Lastly, our work is also different from the recent work that combines the TDP and bottom-up enumeration for web data extraction [Raza and Gulwani 2020]. The goal of the work is to automatically synthesize a program that extracts useful information in a structured format (e.g., a sequence of HTML nodes) from the web. It uses bottom-up enumeration to generate programs that reveal nodes "uniformly distributed" in various aspects (called alignment patterns), independent of any user-provided examples. This information is used to guide the TDP with a few examples to find the desired solution that can generalize well beyond the examples. Though this combination is also synergistic, the major limitation of the TDP – lack of general applicability – still remains. In our work, the bottom-up enumeration allows the TDP to handle arbitrary SyGuS grammars.

Decomposing Synthesis Problems. Our method can be used for a broad class of SyGuS problems with arbitrary grammars, whereas the previous deductive methods to decompose a synthesis problem into subproblems are often limited to special cases. Myth [Osera and Zdancewic 2015] and λ^2 [Feser et al. 2015] use a set of deduction rules for generating subproblems, but they are only applicable to higher-order combinators such as `map` and `filter`. Huang et al. [Huang et al. 2020] have recently proposed three divide-and-conquer strategies. For example, they first try to synthesize subexpressions that appear in a given specification (or, ones that only partially satisfy the specification) and then solve the original synthesis problem using the synthesized subexpressions. Because their method leverages a given complete logical specification, it is not applicable to PBE tasks where a complete specification is missing.

Version space algebra-based techniques [Gulwani 2011; Polozov and Gulwani 2015] also decompose specifications using deductive search, but they limit the expressiveness power of an underlying

DSL to avoid infinitely many decomposed subproblems as explained in Section 1. Furthermore, our work is more beneficial in handling a large number of input-output examples. Our witness functions maintain a set of multiple input-output examples and perform simultaneous decomposition on the examples. On the other hand, the previous approach constructs a VSA for each input-output example, and intersects the multiple VSAs afterward. The intersection-based approach does not scale well in the number of examples because the complexity of VSA intersection is quadratic, which may not be negligible in practice. [8] However, our method is not very sensitive to the number of examples. [9] Another popular divide-and-conquer approach [Alur et al. 2017] finds smaller expressions that are correct on subsets of inputs, predicates that distinguish these subsets, and combines the expressions and predicates to obtain an expression that is correct on all inputs. Our method further improves this strategy as we have shown in Section 4.2.

Bidirectional Enumerative Search. There is a similarity between our method and the *bidirectional enumerative search* used for synthesizing geometry constructions [Gulwani et al. 2011] and superoptimization of assembly code [Phothilimthana et al. 2016]. Given a specification $\varphi \equiv (\varphi_{pre}, \varphi_{post})$ specifying the set of input and output states, the bidirectional enumerative search algorithm maintains two sets of expressions $\widetilde{F}$ and $\widetilde{B}$. The set $\widetilde{F}$ contains expressions obtained by performing a *forward* enumerative search starting from the input states φ_{pre}. The set $\widetilde{B}$ has expressions derived by performing a *backward* enumerative search starting from the output states φ_{post}. It iteratively builds the two sets in order of increasing size until it finds expressions $f \in \widetilde{F}$ and $b \in \widetilde{B}$ such that the states corresponding to f and b can be matched.

Both our method and the bidirectional enumerative search aim for a goal-directed search for solutions. However, a synergistic combination of the two search methods is missing in the bidirectional enumerative search. The forward and backward enumerative algorithms are performed separately, and one does not guide the other. In our method, however, the bottom-up enumeration guides the top-down propagation.

7 CONCLUSION

We presented a general approach to synergistically combine the top-down propagation and bottom-up enumeration. The top-down propagation enables scalable inductive synthesis, but it has been limited to specific application domains. We bring the power of the top-down propagation to SyGuS problems with arbitrary grammar by leveraging the bottom-up enumerative search, enabling a more scalable and general-purpose synthesis strategy. We demonstrated the effectiveness of the approach on a large number of synthesis problems from various application domains. The experimental results show that our method outperforms existing general-purpose and domain-specific synthesis tools.

ACKNOWLEDGMENTS

We thank the reviewers for insightful comments. We are also grateful to Hangyeol Cho for his help in conducting the experiments. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2020R1C1C1014518) and the research fund of Hanyang University (HY-2020-2474).

[8] For this reason, FlashFill/ FlashMeta is often used in a way that examples are provided in an incremental manner (e.g., [Wang et al. 2017] and [Raza and Gulwani 2020]).

[9] Duet performs well in the Bitvec benchmarks each of which comprises up to 1,000 examples. As another example, when solving the benchmark `phone-3-long.sl` in a setting where 100 examples are provided at once, Duet takes 0.1 sec whereas FlashFill takes several minutes.

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD '13)*.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*.

Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard Version 2.0.

Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? arXiv:1510.08393 [cs.LO]

Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press. https://doi.org/10.1017/CBO9780511790492

Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *28th International Conference on Computer Aided Verification (CAV '16)*.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. 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) *(PLDI'15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*.

Sumit Gulwani. 2016. Programming by Examples (and its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press. https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/

Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing Geometry Constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/1993498.1993505

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI'2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. https://doi.org/10.1145/3385412.3385979

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (Cape Town, South Africa) *(ICSE '10)*.

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 776–783.

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*.

DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 503–518. https://doi.org/10.1145/3385412.3385996

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. *SIGPLAN Not.* 53, 4 (June 2018), 436–449. https://doi.org/10.1145/3296979.3192410

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*.

Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443 – 453. https://doi.org/10.1016/0022-2836(70)90057-4

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*.

Past SyGuS Competition. 2020. https://sygus.org/comp/.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS'16)*. Association for Computing Machinery, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction Using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1967–1978. https://doi.org/10.1145/3318464.3380608

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151