Encodings for Enumeration-Based Program Synthesis

Pedro Orvalho[1,2], Miguel Terra-Neves[1,2], Miguel Ventura[2], Ruben Martins[3(✉)], and Vasco Manquinho[1]

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{pmorvalho,vmm}@sat.inesc-id.pt
[2] OutSystems, Lisbon, Portugal
{miguel.neves,miguel.ventura}@outsystems.com
[3] Carnegie Mellon University, Pittsburgh, USA
rubenm@cs.cmu.edu

Abstract. Program synthesis is the problem of finding a program that satisfies a given specification. Most program synthesizers are based on enumerating program candidates that satisfy the specification. Recently, several new tools for program synthesis have been proposed where Satisfiability Modulo Theories (SMT) solvers are used to prune the search space by discarding programs that do not satisfy the specification.

The size of current tree-based SMT encodings for program synthesis grows exponentially with the size of the program. In this paper, a new compact line-based encoding is proposed that allows a faster enumeration of the program space. Experimental results on a large set of query synthesis problem instances show that using the new encoding results in a more effective tool that is able to synthesize larger programs.

Keywords: Program synthesis · Satisfiability Modulo Theories · Enumerative search · SQL

1 Introduction

The goal of program synthesis is to automatically generate programs that satisfy a given high-level specification. Once considered a utopian dream, the recent advances in program synthesis are making this approach more practical and have shown that it can be useful to both end-users and programmers. A common approach is to use input-output examples as specifications. Even though these specifications are incomplete, i.e. a program may satisfy the specification but may not be the program that the user desires, these are easy to create and can be used to solve many real-world applications. This approach is known as programming-by-example (PBE) and has been used to automate tedious tasks in a plethora of applications, such as string manipulations in spreadsheets [10, 15], list transformations [2,9], table reshaping [7], code completion [14], helping programmers to use libraries [8], and SQL queries [18–20]. Program synthesis is

T. Schiex and S. de Givry (Eds.): CP 2019, LNCS 11802, pp. 583–599, 2019.
https://doi.org/10.1007/978-3-030-30048-7_34

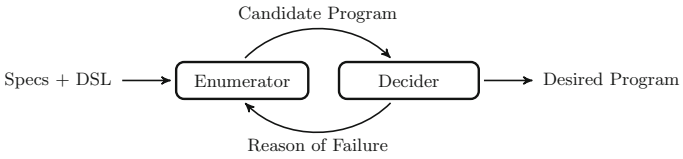


Fig. 1. Enumeration-based program synthesis

not merely an academic research topic since it is also transitioning into industry. Microsoft's FlashFill [10] is the most successful application of program synthesis by Microsoft for string manipulation and it is integrated into Microsoft Excel. Other companies are also starting to look for applications of program synthesis to their products, namely OutSystems [1] and query synthesis.

Even though there are many approaches to program synthesis, the most common one is to perform an enumerative search over the space of programs that satisfy the specifications. Figure 1 shows the high-level architecture of enumeration-based program synthesizers. They take as input the specification that describes the intention of the user (e.g., input-output examples) and a domain-specific language (DSL) that defines the search space. Program synthesizers typically enumerate programs in increasing order of the number of DSL components. For each candidate program $\mathcal{P}$, they check if $\mathcal{P}$ satisfies the specifications. If this is the case, then the desired program was found. Otherwise, the program synthesizer learns a reason for failure and enumerates the next candidate program.

Recent approaches combine enumerative search with deduction with the goal of performing early pruning of infeasible programs [7], or to learn from past failed candidate programs in order to prune all equivalent infeasible programs [6].

Suppose that a user wants to synthesize an SQL query using examples. In particular, given tables *supplier* and *parts* with the schema "supplier(id: integer, sname: string, address: string)" and "parts(id: integer, pname: string, color: string)", the user wants to find the *names* of parts, *pnames*, for which there is some supplier. [1] This could be accomplished with the following SQL query:

```
SELECT pname
FROM parts, supplier
WHERE parts.id = supplier.id
```

To enumerate the space of programs that satisfy the specifications, program synthesizers must first construct an underlying representation of the feasible space. Figure 2 shows the typical tree representation used by program synthesizers (e.g., [3,6,7]), for the above query example. Each node can be a library component or a terminal symbol. Program synthesizers can then traverse the space of possible candidates by enumerating all possible trees of a given depth. However, for approaches that rely on logical deduction, the space of feasible programs must be encoded a priori by using either a Boolean Satisfiability (SAT)

[1] This corresponds to exercise 5.2.1 from a classic textbook on databases [13].

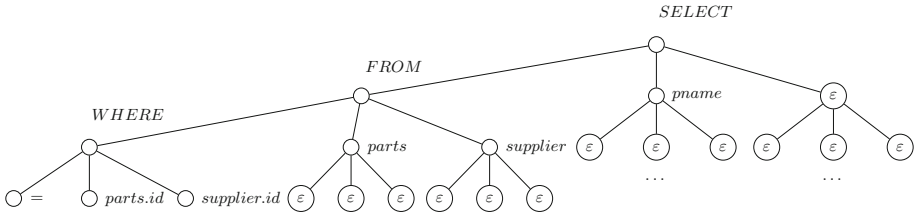


Fig. 2. Tree-based representation of the search space

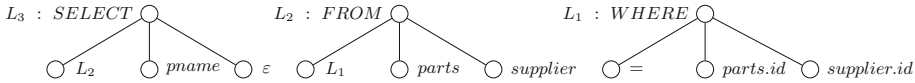


Fig. 3. Line-based representation of the search space

or Satisfiability Modulo Theory (SMT) encoding [6,7]. A common approach to encode all feasible programs is to represent them using a k-tree, where each node has exactly k children and k is the largest number of parameters of the functions in our library of components. Figure 2 shows an example of a 3-tree where each node has 3 children. A complete program corresponds to assigning a label to each node. Components that may have less than 3 parameters (e.g., `SELECT`), will have the empty label *empty* ε assigned to their unused children.

A large downside of a k-tree representation is the exponential growth of the size of the tree with respect to its depth. Since the encoding's complexity depends on the number of nodes, this makes it intractable to enumerate the search space of candidate programs using an SMT encoding.

In this paper, we propose a new line representation illustrated in Fig. 3, where we represent each line with its own subtree and add additional constraints to connect the multiple subtrees. For the above SQL query, we would only need 12 nodes using a line-based representation instead of the 3-tree representation's 40 nodes. When considering programs with 10 lines of code and $k = 4$, the line-based representation only needs 50 nodes instead of the 1,398,101 nodes required by the tree-based representation.

To summarize, this paper makes the following contributions:

- We formalize how to encode the traditional tree-based representation of a program into SMT which has an exponential growth with respect to the number of lines of a program.
- We propose a new compact SMT encoding based on a line representation of programs that grows linearly with the number of lines of a program.
- We integrate the line-based encoding into a program synthesizer and empirically evaluate our approach using SQL benchmarks. Experimental results show that the line-based encoding significantly outperforms the tree-based encoding and allows program synthesizers to more effectively enumerate the search space and synthesize larger programs.

$$
\begin{array}{ll}
table & \rightarrow select\_from(cols,\ table) \mid join(table,\ table) \mid parts \mid supplier \\
cols & \rightarrow column(col) \mid columns(col,\ cols) \\
col & \rightarrow pname \mid sname \mid id \mid color \mid address \mid * \\
empty & \rightarrow empty
\end{array}
$$

Fig. 4. The grammar of a simple DSL for query synthesis; in this grammar, *table* is the start symbol. All joins are natural joins between columns with the same name. Given as input the tables *supplier* and *parts*, with the schema "supplier(id: integer, sname: string, address: string)" and "parts(id: integer, pname: string, color: string)".

2 Preliminaries

The Satisfiability Modulo Theories (SMT) problem is a generalization of the well-known Propositional Satisfiability (SAT) problem. Given a decidable first-order theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$. A $\mathcal{T}$-literal is either a $\mathcal{T}$-atom t or its complement $\neg t$. A $\mathcal{T}$-formula is similar to a propositional formula, but a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals instead of propositional literals. Given a $\mathcal{T}$-formula ϕ, the SMT problem consists of deciding if there exists a total assignment over the variables of ϕ such that ϕ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, among others.

Program synthesizers search the space of programs described by a given domain-specific language (DSL). The syntax of the DSL is described by a context-free grammar G. In particular, G is a tuple (Σ, R, S), where Σ represents the set of symbols, productions, and start symbol, respectively. Each symbol $\sigma \in \Sigma$ corresponds to built-in DSL constructs (e.g., *select_from*, *join*), constants, variables or inputs of the system. Each production rule $p \in R$ has the form $p = (A \rightarrow \sigma(A_1, \ldots, A_m))$, where $\sigma \in \Sigma$ is a DSL construct and $A_1, \ldots, A_m \in \Sigma$ are symbols for the arguments of σ.

Example 1. Consider a DSL D in Fig. 4, and suppose that a user wants to solve the query presented in Sect. 1, i.e. she wants to find all the names of *parts* for which there is some supplier. The desired query from D is the following *select_from(column(pname), join(parts, supplier))*. This query uses three production rules $p_1 =$ *select_from*, $p_2 =$ *column*, and $p_3 =$ *join*; the column *pname*; and input tables *parts* and *supplier*.

3 Tree-Based Encoding

This section describes the tree-based encoding used on several state-of-the-art synthesizers to perform program enumeration. Given a DSL, program synthesis frameworks search for a program that is consistent with the input-output examples provided by the user. For the search process to be complete, these frameworks use a structure capable of representing every possible program up to some given depth of n. Let k be the greatest arity among DSL constructs. For programs with $n-1$ production rules, synthesizers adopt a tree structure of depth n, referred to

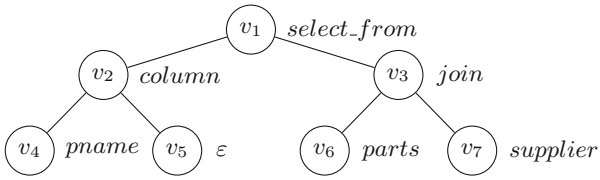


Fig. 5. k-tree representation of the query presented in Example 1

as k-tree, where each node has exactly k children. Figure 5 illustrates a 2-tree that can be used to represent the query presented in Example 1.

In order to perform program enumeration using the tree representation, program synthesizers encode the tree as an SMT formula such that a solution of the SMT formula encodes a complete program by assigning a symbol to each node.

A detailed description of the SMT model follows. First, the encoding variables are introduced. Next, the constraints of the SMT model are presented.

3.1 Encoding Variables

Let s be the length of the DSL's set of symbols, $s = |\Sigma|$. Let $id : \Sigma \rightarrow \mathbb{N}_0$ be a function that maps each symbol to a unique non-negative integer in a one-to-one mapping. As a result, this function provides a unique identifier (integer value between 0 and s) to each symbol in Σ. In our encoding, we assume that the empty production symbol (ε) is mapped to 0 (i.e. $id(\varepsilon) = 0$).

Consider the encoding for a program with a k-tree of depth n. Assume each node in the k-tree is assigned a unique index. Let N be the set of all k-tree nodes indexes such that $N = I \cup L$ where I denotes the set of internal node indexes and L denotes the set of leaf node indexes. Let $C(i)$ denote the set of child indexes of node $i \in N$. Clearly, if i is a leaf node ($i \in L$), then $C(i) = \emptyset$.

In our encoding we define the following variables:

- $V = \{v_i : 1 \leq i \leq |N|\}$: each variable v_i denotes the symbol identifier in node i of the k-tree;
- $B = \{b_i : 1 \leq i \leq |N|\}$: each variable b_i is a Boolean variable that denotes if node i is associated to a production symbol (true) or a terminal symbol (false).

3.2 Constraints

Let D be a DSL, $Prod(D)$ denotes the set of production rules in D and $Term(D)$ the set of terminal symbols in D. Furthermore, let $Types(D)$ denotes the set of types used in D and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ denotes the return type of production rule s.

To ensure that every program enumerated is well-typed the following constraints must be satisfied.

Leaf Nodes. The leaf nodes can only be assigned to terminal symbols because they have no children. Therefore, we define the following constraint:

$$\forall i \in L : \bigvee_{p \in Term(D)} v_i = id(p) \tag{1}$$

Example 2. Given the DSL D from Fig. 4, the set of terminal symbols is $Term(D) = \{parts, supplier, pname, sname, id, color, address, *, \varepsilon\}$ and the set of leaves is $L = \{4, 5, 6, 7\}$. Each leaf node in L must be assigned to a symbol in $Term(D)$. Hence, each leaf $i \in L$ must satisfy: $v_i = id(parts) \vee v_i = id(supplier) \vee v_i = id(pname) \vee v_i = id(sname) \vee v_i = id(id) \vee v_i = id(color) \vee v_i = id(address) \vee v_i = id(*) \vee v_i = id(\varepsilon)$.

Internal Nodes. If a production rule p is assigned to an internal node, then the type of its children nodes must match the types of parameters of p. Let $Type(p, j)$ denote the type of parameter j of production rule $p \in Prod(D)$. If $j > arity(p)$, then $Type(p, j) = empty$. If p is a terminal symbol, $p \in Term(D)$, then for every j, $Type(p, j) = empty$.

Let $\Sigma(Type(p, j))$ represent the subset of symbols in Σ of type $Type(p, j)$.

$$\forall i \in I,\ j \in C(i),\ p \in \Sigma\ : \\ v_i = id(p) \Rightarrow \bigvee_{t \in \Sigma(Type(p,j))} v_j = id(t) \tag{2}$$

With constraint (2), all the programs generated will be well-typed since each node is only assigned to a production rule if its children have the correct type.

Example 3. Consider again the query in Example 1. If the production *select_from* is assigned to the program's root, v_1, then $\Sigma(Type(select\_from, 1)) = \{column, columns\}$ and $\Sigma(Type(select\_from, 2)) = \{select\_from, join, parts, supplier\}$. The following constraint must be satisfied: $v_1 = id(select\_from) \Rightarrow \big(v_2 = id(column) \vee v_2 = id(columns)\big)$ and $v_1 = id(select\_from) \Rightarrow \big(v_3 = id(select\_from) \vee v_3 = id(join) \vee v_3 = id(parts) \vee v_3 = id(supplier)\big)$.

Output. Let t be the output type. Furthermore, consider that the program root identifier is 1. Then, v_1 must be assigned to a symbol that is consistent with the output type t. Hence, the following constraint must be satisfied.

$$\bigvee_{s \in \Sigma(t)} v_1 = id(s) \tag{3}$$

Input. Let IN be the set of symbols provided by the user as input. In order to guarantee that all generated programs use all the inputs provided by the user, the following constraint is added:

$$\forall p \in IN : \bigvee_{i \in N} v_i = id(p) \tag{4}$$

$$
\begin{aligned}
L_1 &: \; ret_1 \;\leftarrow\; column(pname) \\
L_2 &: \; ret_2 \;\leftarrow\; join(parts,\; supplier) \\
L_3 &: \; ret_3 \;\leftarrow\; select\_from(ret_1,\; ret_2)
\end{aligned}
$$

Fig. 6. Line-representation of the query from Example 1

Note that this is not required for the encoding's correction. Nevertheless, we are only interested in enumerating programs that use all inputs given by the user.

Exactly $n-1$ ***Production Rules.*** Finally, we are interested in enumerating programs using Exactly $n-1$ production rules by adding the following constraints:

$$\forall i \in N : b_i = 1 \iff \bigvee_{p \in Prod(D)} v_i = id(p) \tag{5}$$

$$\Big(\sum_{i \in N} b_i\Big) = n - 1 \tag{6}$$

With constraints (5) and (6), we guarantee that given a k-tree of depth n, each enumerated program will have exactly $n-1$ production rules. State-of-the-art program synthesizers iteratively search for programs in increasing depth. Thus, constraint (6) allows pruning the search space, in order to avoid enumerating repeated programs in future iterations of depth greater than n.

Encoding Complexity. Let k be the greatest arity between DSL constructs and let n denote the number of productions (lines of code) in a program. In terms of nodes complexity, the number of nodes increases exponentially with the number of productions, as follows: $\frac{k^{n+1}-1}{k-1}$.

4 Line-Based Encoding

In this section, we propose a new encoding to represent programs. Our goal is to represent a program as a sequence of lines where each line represents an operation in the DSL. Instead of using a single k-tree to represent a program, each line is represented as a tree with a depth of 1.

Consider the program in Fig. 6. One can represent this program as three trees of depth 1 as shown in Fig. 7. Note that the result of the program is the value returned by the third tree. Observe that ret_i is a new symbol that represents the return value of line i.

4.1 Encoding Variables

Recall that D denotes a DSL, $Prod(D)$ the set of production rules in D and $Term(D)$ the set of terminal symbols in D. Furthermore, $Types(D)$ denotes the set of types used in D and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ denotes the return type of production rule s.

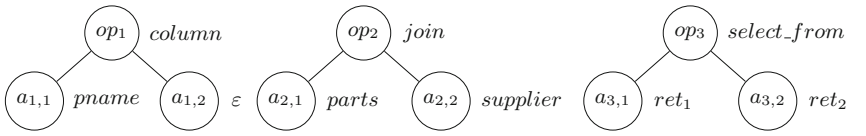


Fig. 7. Each tree represents a production rule. The first tree represents line 1, the second tree represents line 2 and the third tree represents line 3. ret_1 (resp. ret_2) denotes the value returned in line 1 (resp. line 2).

Consider the encoding for a program with n lines where the maximum arity of the operators is k, then we have the following variables:

- $O = \{op_i : 1 \leq i \leq n\}$: each variable op_i denotes the production rule used in line i;
- $T = \{t_i : 1 \leq i \leq n\}$: each variable t_i denotes the return type of line i;
- $A = \{a_{ij} : 1 \leq i \leq n, 1 \leq j \leq k\}$: each variable a_{ij} denotes the symbol corresponding to argument j in line i.

4.2 Constraints

Besides the production rules $Prod(D)$ and terminal symbols $Term(D)$, we define one return symbol for each line in the program. Let $Ret = \{ret_i : 1 \leq i \leq n\}$ denote the set of return symbols in the program.

In our encoding, we define a different non-negative identifier for each symbol. Here, we extend the id function to also consider the symbols that represent the return value of each line. Let $Symbols = Prod(D) \cup Term(D) \cup Ret$ define the set of all symbols used in the program. Finally, let $id : Symbols \rightarrow \mathbb{N}_0$ and $tid : Types(D) \rightarrow \mathbb{N}_0$ be one-to-one mappings of symbols and types, respectively, to non-negative integer values.

Operations. First, the operations in each line must be production rules. Hence, we have the following set of constraints:

$$\forall 1 \leq i \leq n : \bigvee_{p \in Prod(D)} (op_i = id(p)) \tag{7}$$

The operation symbol used in each line implies the line's return type.

$$\forall 1 \leq i \leq n, p \in Prod(D) : (op_i = id(p)) \Rightarrow (t_i = tid(Type(p))) \tag{8}$$

Given a sequence of operations, the arguments of operation i must either be terminal symbols or return symbols from previous operations. Hence, we have:

$$\forall 1 \leq i \leq n, 1 \leq j \leq k : \bigvee_{s \in Term(D) \cup \{ret_r : r < i\}} (a_{ij} = id(s)) \tag{9}$$

Arguments. The arguments for a given operation i must have the same types as the parameters of the production rule used in the operation. Let $Type(p, j)$

denote the type of parameter j of production rule $p \in Prod(D)$. If $j > arity(p)$, then $Type(p, j) = empty$. Hence, we have the following constraints when a return symbol is used as an argument of an operation:

$$\begin{aligned} \forall 1 \leq i \leq n, p \in Prod(D), 1 \leq j \leq arity(p), 1 \leq r < i : \\ ((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \Rightarrow (t_r = tid(Type(p, j))) \end{aligned} \tag{10}$$

A given terminal symbol $t \in Term(D)$ cannot be used as argument j of an operation i if it does not have the correct type:

$$\begin{aligned} \forall 1 \leq i \leq n, p \in Prod(D), 1 \leq j \leq arity(p), \\ s \in \{t \in Term(D) : Type(t) \neq Type(p, j)\} : \\ (op_i = id(p)) \Rightarrow (a_{ij} \neq id(s)) \end{aligned} \tag{11}$$

Since the arity of a given operation i can be smaller than k, we must also have that the arguments above the production's arity must be assigned to the empty symbol ε:

$$\begin{aligned} \forall 1 \leq i \leq n, p \in Prod(D), arity(p) < j \leq k : \\ (op_i = id(p)) \Rightarrow (a_{ij} = id(\varepsilon)) \end{aligned} \tag{12}$$

Output. Let $Type(output)$ denote the type of the program's output and let $P_O \subseteq Prod(D)$ be the subset of production rules with return type equal to Type(output), i.e., $P_O = \{p \in Prod(D) : Type(p) = Type(output)\}$. The following constraint ensures that the program's output (last line, n^{th}) has the desired type.

$$\bigvee_{p \in P_O} (op_n = id(p)) \tag{13}$$

Input. Let IN be the set of symbols provided by the user as input. In order to guarantee that all generated programs use all the inputs, the following constraint is used:

$$\forall s \in IN : \bigvee_{1 \leq i \leq n} \bigvee_{1 \leq j \leq k} (a_{ij} = id(s)) \tag{14}$$

Lines Used Exactly Once. A feature of this new encoding is that the result of a given operation can be used more than once. Notice that in the tree-based encoding, one would have to reproduce the same operations in a different branch of the tree. In order to compare the two types of enumeration, tree-based and line-based, we can add a set of constraints restricting the usage of each operation's result to only one usage. Clearly, the following constraints are not necessary to the encoding's correction.

$$\forall ret_r \in Ret(D) : \left(\sum_{r < i \leq n, 1 \leq j \leq k} (a_{ij} = id(ret_r)) \right) = 1 \tag{15}$$

$$
\begin{array}{lll}
L_1 & : ret_1 & \leftarrow column(pname) \\
L_2 & : ret_2 & \leftarrow join(parts,\ supplier) \\
L_3 & : ret_3 & \leftarrow select\_from(ret_1,\ ret_2)
\end{array}
\qquad
\begin{array}{lll}
L_1 & : ret_1 & \leftarrow join(parts,\ supplier) \\
L_2 & : ret_2 & \leftarrow column(pname) \\
L_3 & : ret_3 & \leftarrow select\_from(ret_2,\ ret_1)
\end{array}
$$

Fig. 8. Two line representations of the program from Example 1

Encoding Complexity. Let k be the greatest arity between DSL constructs and let n denote the number of productions (lines of code) in a program. In terms of nodes complexity, we can observe a drastic difference between both types of enumeration, tree-based and line-based. In tree-based enumeration, the number of nodes increases exponentially with the number of productions. On the other hand, the number of nodes used by line-based enumeration increases linearly, $(k+1) \times n$, because the enumerator uses n trees, with $k+1$ nodes each, to represent a program with n production rules.

4.3 Symmetric Programs

In line-based encoding, the number of solutions of the SMT formula is larger than the number of solutions in the corresponding tree-based encoding. There are two main reasons for this difference: (1) the line-based encoding can use the return value of the same line of code more than once, and (2) the same program can have more than one representation, i.e. symmetric programs.

Regarding reason (1), with constraint (15), we guarantee that the return value of each line is used exactly once. Concerning reason (2), in the line-based encoding, some programs can be represented with different sequences of lines. However, in the tree-based encoding, as a result of the single tree representation, the arguments of each production rule will always come from the same branch.

Example 4. Consider the DSL in Fig. 4 and the program *select_from(column (pname), join(parts, supplier))* from Example 1. In tree-based encoding this program has a single representation shown in Fig. 5. However, for the same program, line-based encoding has two possible representations shown in Fig. 8.

In order for the line-based process to enumerate the same number of solutions than the tree-based enumeration, it is necessary to find the symmetries in the line-based encoding and block them. Otherwise, symmetric programs as the one in Fig. 8 will be enumerated and the synthesizer will have to check both programs. Therefore, if we have a solution α of line-based SMT formula and the synthesizer verifies that the corresponding program is not consistent with the input-output examples, then all solutions that encode symmetric programs in relation to α can be blocked.

A simple way to find these symmetries is through a directed acyclic graph of dependencies, where a vertex is defined for each program line, and edges correspond to the line dependencies in the program. Let v_i and v_j denote the vertexes in the graph corresponding to lines i and j with $i < j$. If the return

value of line i is used as argument in line j, then a directed edge (v_i, v_j) must be added to the graph. After building the graph, one can enumerate all possible topological orders of vertexes in the dependency graph. Next, each program associated with a topological order is blocked in the SMT formula.

Example 5. Consider the program from Example 1. Line 3 (L_3) depends on line 1 (L_1) and line 2 (L_2). Therefore, lines 1 and 2 must occur before line 3. However, the order of lines 1 and 2 can be changed. Hence, two solutions would be blocked corresponding to permutations $L_1 - L_2 - L_3$ and $L_2 - L_1 - L_3$ of the program.

5 Experimental Results

In order to evaluate the new line-based encoding, we integrated our proposal in the Trinity [4] synthesis framework. By default, Trinity uses tree-based enumeration to search for programs and uses the Z3 SMT solver [5] with the theory of Linear Integer Arithmetic in the enumeration process. Trinity, like most PBE state-of-the-art synthesizers, takes as input a DSL, a set of examples, and any constants or aggregate functions (e.g., `mean`) that the query may need. Trinity starts by searching for programs with 1 production rule and iteratively increases this bound until a program that satisfies all input-output examples is found.

All of the experiments presented in this section were conducted on an Intel(R) Xeon(R) with E5-2630 v2 2.60 GHz CPUs, using a memory limit of 64 GB and a time limit of 3,600 s. The goal of our evaluation was to answer the following questions:

Q1. How does line-based enumeration compare against tree-based enumeration in terms of encoding complexity? (Sect. 5.2)

Q2. How does line-based enumeration compare against tree-based enumeration in general? (Sect. 5.2)

Q3. How does line-based enumeration compare against tree-based enumeration for programs with more than three lines of code? (Sect. 5.2)

Q4. What is the performance impact of breaking symmetries in line-based enumeration? (Sect. 5.3)

5.1 SQL Benchmarks

We designed a DSL for SQL that can solve classic SQL queries from a database textbook [13]. These benchmarks were previously used by well-known SQL synthesizers [7,18,20]. We started with an initial set of 23 SQL benchmarks (corresponding to Sects. 5.1.1 and 5.1.2 of the database textbook [13]) and created variants of these benchmarks until a total of 55 benchmarks.

Since we want to study the performance of each encoding with respect to the size of the synthesized query, for each of these benchmarks, we generate six different SMT formulas to search for programs that use exactly n production rules from our DSL, for a total of 330 benchmarks (55×6). The SMT formulas differ in the number of productions that their programs must have, and it simulates the search performed by a program synthesizer until a solution is found.

Table 1. Number of tree nodes, variables and mean number of constraints used by each approach for a given program's size.

	Encoding					
	Tree-based			Line-based		
Lines of code	Nodes	Variables	Constraints	Nodes	Variables	Constraints
1	5	10	379	5	6	44
2	21	42	1,703	10	16	118
3	85	170	6,999	15	30	224
4	341	682	28,183	20	48	362
5	1,365	2,730	112,919	25	70	532
6	5,461	10,922	451,863	30	96	734

Table 2. Number of solved benchmarks by each approach.

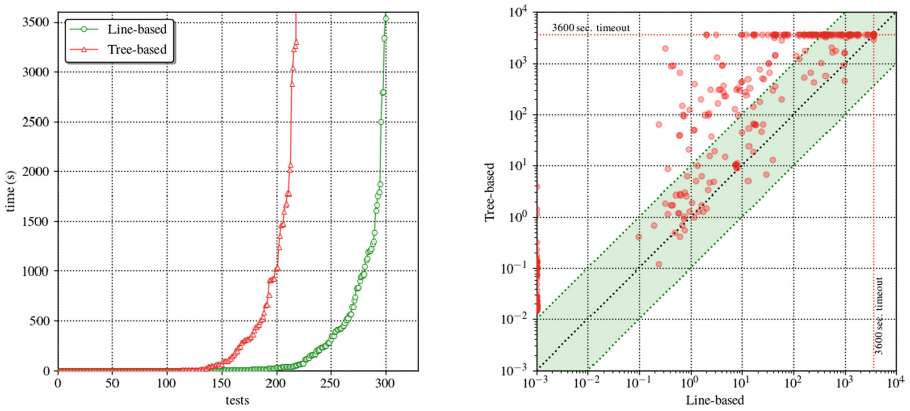
Lines of code	1	2	3	4	5	6	Total	% Solved	% Solved (LOC >= 4)
# Tests	55	55	55	55	55	55	330		
Tree-based	55	55	54	34	18	2	218	66.06%	32.73%
Line-based	55	55	54	49	48	39	300	90.91%	82.42%

5.2 Comparison Between Line-Based and Tree-Based Encodings

Encoding Complexity. As presented in Sects. 3 and 4, the number of nodes used by line-based enumeration increases linearly. On the other hand, in tree-based enumeration the number of nodes increases exponentially with the number of productions. The number of variables and constraints used by each type of enumeration varies with the number of nodes. Table 1 shows the number of nodes, variables and the mean number of constraints used by each type of enumeration on the 330 SQL benchmarks. The number of nodes and variables are always the same for a given program's size. The number of constraints varies with the DSL since each benchmark may use different constants and aggregate functions.

Performance. Table 2 shows the number of solved benchmarks by each encoding for a given number of lines in our DSL. The performance for both encodings is similar for programs with three or fewer lines of code. However, when the program size increases, the difference between these approaches becomes clear. The last line of Table 2, shows the percentage of solved benchmarks by each approach with more than three lines of code. The tree-based encoding only solves 33% of the benchmarks while line-based encoding solves 82%.

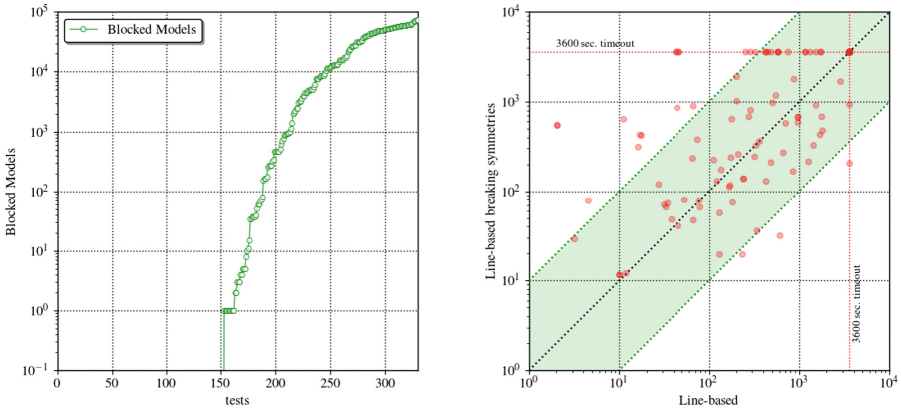
In terms of time spent in each benchmark, Fig. 9 shows two plots, a cactus plot in Fig. 9a and a scatter plot in Fig. 9b. The cactus plot shows the cumulative synthesis time (y-axis) against the number of benchmarks solved (x-axis). Each point in the scatter plot represents a benchmark where the x-value (resp. y-value) is the time spent by the line-based (resp. tree-based) enumerator on



(a) Running times.

(b) Comparison between encodings.

Fig. 9. Tree-based vs Line-based Enumerators.



(a) Number of blocked symmetric solutions per benchmark.

(b) Runtime comparison without considering the overhead to find symmetries.

Fig. 10. Impact of breaking symmetries.

a given benchmark. Both plots, in Fig. 9, support the results shown in Table 2. Additionally, the plots show that tree-based enumeration is, in general, significantly slower than line-based enumeration.

These differences in time and number of benchmarks solved, in particular for the instances with more than 3 lines, can be justified by the exponential number of variables and constraints required by tree-based enumeration.

5.3 Impact of Symmetry Breaking

We evaluate the impact of symmetry breaking on the performance of line-based enumeration. For every solution α, we find all solutions symmetric to α and add constraints to block them. Our experiments show that symmetry breaking does not improve the performance of line-based enumeration. Possible explanations for this behavior can be: (1) the number of symmetries is only significant for programs with more than three lines, and (2) the overhead to find and block all symmetric solutions is too large when compared to the time of each SMT call.

Figure 10a shows the total number of symmetric solutions blocked in each benchmark using a logarithmic scale. Programs with one or two lines of code do not have symmetries because they have only one representation. Programs with three lines of code have at most one symmetry. Therefore, only programs with more than three lines of code, have a significant number of blocked solutions, i.e., blocked more than a thousand symmetric solutions (117 benchmarks). If we only look at these 117 benchmarks, we observe that not breaking symmetries solves 87 benchmarks, while breaking symmetries only solves 68 benchmarks.

Since breaking symmetries is ineffective even when a large number of symmetries is present, we analyzed the current overhead of finding and blocking symmetric solutions. For each solution, we spend on average 0.091 s to find and block all symmetric solutions. Figure 10b shows, per benchmark, the time spent by the line-based enumerator with and without symmetry breaking, ignoring the time spent searching for and blocking symmetric solutions. This plot shows that, even if symmetry breaking was free, it does not improve the performance of the line-based enumerator. We observed that, without symmetry breaking, each SMT call takes on average 0.015 s. If we add symmetry breaking predicates, each SMT call doubles its time to 0.030 s, on average. Since our enumeration relies on solving many easy SMT calls, we concluded that the search space reduction enabled by symmetry breaking does not compensate for the extra effort required to break symmetries.

6 Related Work

Program synthesis has been widely used to synthesize queries using input-output examples [4,7,17,18,20] or natural language [19]. Approaches for query synthesis vary from using decision trees with fixed templates [17,20], to abstract representations of queries that can potentially satisfy the input-output examples [18], and to use SMT-based over-approximations to prune the search space [7].

Tree representations of program search spaces are commonly used in modern program synthesis applications. For example, Bonsai [3] is a validation framework for type systems that uses such representations to synthesize syntactically incorrect programs wrongly accepted by the type checker. State-of-the-art program synthesizers based on enumeration [4,6,7] also make use of tree-based SMT encodings in order to prune the search space by checking if it is possible to extend a given partial program to a complete program which satisfies the input-output examples. The encodings studied in this paper can improve the

enumeration of program synthesizers based on SMT encodings [4,6,7]. These encodings are domain agnostic and can be used in other domains besides SQL (e.g., lists, strings, tables, etc.) with expected improvements of the same order of magnitude.

Alternatively, the synthesis problem can be directly encoded into SMT using quantified formulas [11,12,16]. Brahma [11,12] is an oracle-guided synthesizer that considers an SMT encoding with some similarities to the line-based encoding proposed in Sect. 4. However, there are some fundamental differences: (1) The program specification is generic and must be satisfied for all possible program inputs. Therefore, Brahma essentially solves a single universally quantified SMT formula in order to synthesize a program. (2) SMT specifications must capture the complete semantics of the respective components, while state-of-the-art enumeration-based synthesizers typically deal with specifications that over-approximate the components' behavior. (3) The authors focus on bit-vector manipulation and do not consider arguments of different types. Synudic [16] extends Brahma with additional restrictions on the structure of the program to be synthesized. This allows users to either search for all programs that satisfy the functional requirements or to narrow the search space to programs that satisfy a given template. Even though there are some similarities between our encoding and a purely SMT-based approach [11,12,16], we only need to encode a formula where each solution represents a well-typed program. The SMT encoding abstracts the semantics of the operators and is simpler than previous approaches that encode the entire synthesis problem as an SMT problem. Moreover, an enumeration-based approach makes thousands of SMT calls (each in the order of milliseconds), while the SMT encodings from previous approaches [11,12,16] typically solve one large quantified formula that can take a very long time.

7 Conclusions

In recent years, new platforms for software development have been made available where users with little programming skills are able to create and modify software applications. These tools are able to hide many aspects of programming, but some coding experience is still needed for some operations. Programming-by-example is making programming more accessible by allowing users to create incomplete specifications through input-output examples of these operations and automatically synthesize the desired program.

Currently, the most common approach to program synthesis is to perform an enumerative search on the space of programs and find one that satisfies the specifications. In this paper, we propose a new compact SMT encoding for program enumeration where each program is represented as a sequence of lines. Experimental results on synthesis of SQL queries show that the proposed line-based encoding allows a faster enumeration of programs when compared to the usual tree-based encoding. Moreover, while the tree-based encoding does not scale beyond a small number of operations, the new line-based encoding allows to find programs with a larger sequence of operations.

Acknowledgments. This work was supported by national funds through FCT with references UID/CEC/50021/2019, PTDC/CCI-COM/31198/2017 and DSAIPA/AI/0044/2018, and by NSF award #1762363 and the CMU-Portugal program with reference CMU/AIR/0022/2017.

References

1. OutSystems (2019). https://www.outsystems.com. Accessed 15 July 2019
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: learning to write programs. In: Proceedings International Conference on Learning Representations. OpenReview (2017)
3. Chandra, K., Bodík, R.: Bonsai: synthesis-based reasoning for type systems. PACM Program. Lang. **2**(POPL), 1–34 (2018)
4. Chen, J., Martins, R., Chen, Y., Feng, Y., Dillig, I.: Trinity: an extensible synthesis framework for data science. PVLDB **12**(12), 1914–1917 (2019)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: Proceedings Conference on Programming Language Design and Implementation, pp. 420–435. ACM (2018)
7. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings Conference on Programming Language Design and Implementation, pp. 422–436. ACM (2017)
8. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.: Component-Based Synthesis for Complex APIs. In: Proceedings Symposium on Principles of Programming Languages, pp. 599–612. ACM (2017)
9. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings Conference on Programming Language Design and Implementation, pp. 229–239. ACM (2015)
10. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings Symposium on Principles of Programming Languages, pp. 317–330. ACM (2011)
11. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings Conference on Programming Language Design and Implementation, pp. 62–73. ACM (2011)
12. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings International Conference on Software Engineering, pp. 215–224. ACM (2010)
13. Ramakrishnan, R., Gehrke, J.: Database management systems. McGraw Hill, New York (2000)
14. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings Conference on Programming Language Design and Implementation, pp. 419–428. ACM (2014)
15. Singh, R., Gulwani, S.: Transforming spreadsheet data types using examples. In: Proceedings Symposium on Principles of Programming Languagesm, pp. 343–356. ACM (2016)

16. Tiwari, A., Gascón, A., Dutertre, B.: Program synthesis using dual interpretation. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 482–497. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_33
17. Tran, Q.T., Chan, C., Parthasarathy, S.: Query by output. In: Proceedings International Conference on Management of Data, pp. 535–548. ACM (2009)
18. Wang, C., Cheung, A., Bodik, R.: Synthesizing highly expressive SQL queries from input-output examples. In: Proceedings Conference on Programming Language Design and Implementation, pp. 452–466. ACM (2017)
19. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: SQLizer: query synthesis from natural language. In: Proceedings International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 63:1–63:26. ACM (2017)
20. Zhang, S., Sun, Y.: Automatically synthesizing SQL queries from input-output examples. In: Proceedings International Conference on Automated Software Engineering, pp. 224–234. IEEE (2013)