Program Synthesis from Noisy Tabular Data

Daniel Ramos
daniel.r.ramos@tecnico.ulisboa.pt

INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal

October 2019

Abstract

This document targets program synthesis, the task of automatically generating programs from a set of constraints. Interest in program synthesis has soared in recent years, mainly because of what it promises to deliver: to allow users with little programming knowledge to indirectly program, by simply providing input-output examples. Although recent program synthesizers are capable of synthesizing programs from input-output examples, most of them cannot deal with noise. Therefore, a simple mistake in one of those examples can render the synthesis task unfeasible. Moreover, to the best of our knowledge, the task of synthesizing programs from a set of noisy input-output examples remains unexplored. In this document, we aim to solve program synthesis in such a setting. We divide our contributions into three sections: (1) a hybrid algorithm for program enumeration that uses both SMT and graph-based search; (2) a generalized synthesis algorithm to solve the problem of synthesizing programs from noisy examples; (3) a synthesizer capable of synthesizing programs from noisy input-output examples of tables.
Keywords: Program Synthesis, Satisfiability Modulo Theories, Noisy Examples, Table Transformations

1. Introduction

Suppose we have a list of names with over three thousand entries. From this list we would like to extract the surname of each entry for statistical purposes. From a programmer's point of view, the task seems rather easy. We can simply write a regular expression and match it against each name, using a tool of our choice. However, from a non-programmer's point of view, this task can be a big headache. Despite not having any knowledge about programming, the non-programmer would certainly like to have a way of automating this tedious and repetitive task as well. Ideally, he would verbalize his intent to some kind of assistant and it would return a program to automate this task.

This problem is known as program synthesis: the problem of automatically generating programs in a programming language from a high-level specification, such as a first-order logic formula or input-output examples[13]. In this paper, we focus on the problem of synthesizing programs from input-output examples. Specifically, we seek to synthesize programs from a set of noisy input-output examples of tables that only roughly approximate the desired input-output behavior. Informally, our new problem statement is as follows. Given a set of tables A, a table B obtained from a noisy source, and a programming language $\mathcal{L}$, is it possible to find a program $P \in \mathcal{L}$ such that $P(A) \approx B$?

1.1. Document Structure

This document is organized as follows. In section 2, we start with a brief introduction to program synthesis, where we explain the fundamental concepts related to the different dimensions of program synthesizers. In section 3, we propose a generalized program synthesis algorithm that leverages both SMT and graph-based search. In section 4, we present and discuss the results we obtained from empirically evaluating our algorithm on a set of 487 benchmarks. Finally, the paper concludes in section 5.

2. Background

According to Gulwani et al.[13], a program synthesizer is generally defined across 3 dimensions, (1) the user intent or problem specification, (2) the program space, and (3) the search technique. In this section, each one of these dimensions is introduced in detail.

2.1. Problem Specification

In traditional compilers, there is a well-defined language in which we express the operations we want the computer to do. Similarly, program synthesizers must have a mechanism of understanding their users. The big difference between the two is in the way they perceive the users' intention. Instead of interpreting an artificial programming language like compilers, program synthesizers cap-

ture user intent through mechanisms that are familiar to the average user, that is, it does not require them to learn a new formalism. These mechanisms can range from complete formal specifications such as a first-order formula fully describing the program functionality[9, 10, 12, 23], or ambiguous specifications such as pairs of input-output examples[6, 7, 8, 11, 14, 15, 18, 19, 22, 24, 26] or even natural language[4, 25].

In this work, we focus on Programming by Example (PBE), a branch of program synthesis that uses pairs of input-output examples as the problem specification for the synthesizer.

Definition 1. (Problem specification) The problem specification Λ of the program synthesis problem is a set of n input-output examples $\{(\vec{x}_i, y_i) : 0 \leq i \leq n-1\}$, where $\vec{x}_i$ is the vector with the input values of example i and y_i is the respective output.

2.2. Program Space

The number of programs in a programming language is usually infinite. For example, the *body* of a *while* statement in `C++` can have an arbitrary number of statements of unlimited depth. Naturally, in order to make program synthesis feasible, the space in which synthesizers operate has to be limited. A naive approach would be to place a hard limit on the depth and arity of each operator. In practice, the combinatorial nature of the problem still requires a further reduced search space. This is why program synthesizers usually explore a small subset of a programming language instead of the programming language itself. This subset is known as the program space, and it can be defined by using a domain specific language[13].

Definition 2. (Domain specific language) A domain specific language (DSL) $\mathcal{L}$ is a formal language that can be described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where V is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, R is a finite relation from V to $(V \cup \Sigma)^*$ defining the production rules, and S is the start symbol.

Throughout this work, it is assumed that every grammar used to describe the DSL only contains three types of terminal symbols, the component (function) related symbols, the n program inputs $x_0, x_1, ..., x_{n-1}$, and constants. We represent each production rule of a DSL in form $lhs \rightarrow rhs$, where lhs is the non-terminal symbol that is replaced by the string rhs. In particular, we represent each production rule corresponding to a component by $\mathcal{A}_0 \rightarrow \beta(\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n)$, where β is the component (a terminal symbol), and $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n$ are non-terminal symbols corresponding to its arguments.

Example 1. Assuming x_0 is the only program input, we can define a simple DSL $\mathcal{L}$ in the domain of table manipulation:

$$S \rightarrow \mathit{select}(S, M) \mid \mathit{transpose}(S) \mid x_0$$
$$M \rightarrow 1 \mid 2 \mid 3 \mid ... \mid 10$$

Given a DSL, we can define the notions of complete program and sketch.

Definition 3. (Program) A program P in a DSL $\mathcal{L}$ described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ is a string $P \in (\Sigma \cup V)^*$, such that $S \overset{*}{\Rightarrow} P$.

Definition 4. (Complete program) A complete program P_c in a DSL $\mathcal{L}$ described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ is a string $P_c \in (\Sigma)^*$, such that $S \overset{*}{\Rightarrow} P_c$.

Example 2. Consider the DSL $\mathcal{L}$ defined in Example 1. Some examples of complete programs are $\mathit{select}(x_0, 5)$, $\mathit{transpose}(x_0)$, and $\mathit{select}(\mathit{transpose}(x_0), 1)$. Moreover, $\mathit{select}(S, M)$, $\mathit{transpose}(S)$, and $\mathit{select}(\mathit{transpose}(S), 1)$ are programs, but they are not complete programs because they contain non-terminal symbols.

Similarly to Feng et al.[5], we represent programs as abstract syntax trees (AST), tree representations of the syntactic structure of programs. We think of nodes of an AST as structures with 3 attributes, (1) the DSL symbol it represents, (2) a list of pointers to its children, and (3) a unique index. Given a program P, we refer to the root node of its AST by $\mathrm{Root}(P)$. Given a node N, we use $\mathrm{Ch}(N)$ to denote an ordered list of its children. We use s_N to denote the unique index associated with node N. Given a program P, we associate index $s_R = 0$ with its root node $R = \mathrm{Root}(P)$. The index of the j'th child of node N is $s_N \times b + j$, where b is the maximum arity of any DSL component.

Example 3. Consider the DSL $\mathcal{L}$ defined in Example 1. By successively applying the productions of $\mathcal{L}$ starting from the start symbol, we can derive the program $P := \mathit{select}(\mathit{transpose}(x_0), 1)$, which can be represented by the AST illustrated in Figure 1.

2.3. Search Technique

In the classical version of program synthesis, the synthesizer's objective is to find a program that satisfies every input-output example. This formulation is too narrow for our purposes, since it cannot fit the problem we aim to solve: synthesizing programs from noisy input-output examples. Therefore, we propose an alternative formulation.

Problem statement 1. Given a domain specific language $\mathcal{L}$, a problem specification Λ with

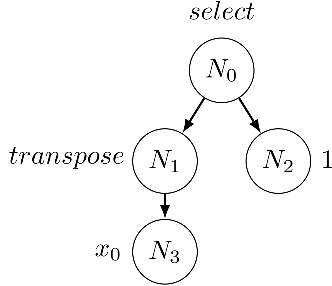


Figure 1: AST representation of the program $select(transpose(x_0), 1)$. N_{s_N} denotes the node with index s_N. The DSL symbol associated with each node is next to it.

n input-output examples $(\vec{x_i}, y_i) \in \Lambda$, and a cost function to measure the quality of programs $c : \mathcal{L} \times S \mapsto \mathbb{R}^+$, where S is the set of all problem specifications, the generalized program synthesis problem is to find a complete program $P \in \mathcal{L}$ such that $P = \text{argmin}_{P' \in \mathcal{L}} c(P', \Lambda)$.

The search technique we use to solve the synthesis problem largely depends on the internal representation we choose. For instance, if we model our problem as state-space search problem, we will most certainly use a graph search algorithm. Whereas if choose to model our problem as set of constraints, we will most likely use a constraint solver. In this paper, we opt for a hybrid representation that uses both graphs and logical formulas. Therefore, we introduce some concepts related to the satisfiability of formulas.

Consider a set $V = \{v_1, v_2, ..., v_n\}$ of n Boolean variables. A literal is a variable $v_i \in V$ or its negation $\neg v_i$. A clause is a disjunction of literals. A propositional formula in conjunctive normal form (CNF) is a conjunction of clauses. Given a propositional formula in CNF φ, the propositional satisfiability (SAT) problem consists in deciding if there exists an assignment to each variable in V such that φ is satisfied. The satisfiability modulo theories (SMT) is a generalisation of SAT, where the domain of variables depends on a given theory $\mathcal{T}$. A $\mathcal{T}$-atom is ground atomic formula in theory $\mathcal{T}$. A $\mathcal{T}$-literal is a $\mathcal{T}$-atom t or its negation $\neg t$. A $\mathcal{T}$-formula is a conjunction, or a disjunction of $\mathcal{T}$-literals. Given a $\mathcal{T}$-formula φ, we say that it $\mathcal{T}$-satisfiable if there exists an assignment to each variable of φ such that φ is satisfied.

Example 4. Consider the following SMT formula in the theory of linear integer arithmetic (LIA):

$$(x + y = 2) \wedge (z \geq x + y) \wedge (x = 1)$$

The assignment $\{x = 1, y = 1, z = 2\}$ serves as proof that the formula is satisfiable.

Recall that our goal is to represent the program space using graphs and logical formulas. In particular, we are mostly interested in using SMT formulas. One of our motivations is that if we encode the program space in SMT, we can use off-the-shelf SMT solvers[1] such as Z3[3] to enumerate programs. Therefore, we do not have to fully design a special propose algorithm to solve our problem.

3. Implementation

In order to build a program synthesis algorithm, we first need to have a way to enumerate programs. Our objective is to divide the enumeration of programs in a loop consisting of two smaller steps: (1) build an incomplete program comprised of only components (a sketch), and (2) complete the sketch in all possible ways until a satisfying program is found or the sketch is exhausted. For clarification purposes we formally define the notion of sketch.

Definition 5. (Sketch) Consider a domain specific language $\mathcal{L}$ described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where $\Sigma = \mathcal{F} \cup \mathcal{I} \cup \mathcal{C}$[2], in which $\mathcal{F}$ is the set of component related symbols, $\mathcal{I}$ is the set of input variable symbols, and $\mathcal{C}$ is the set of constant symbols. A sketch $P \in \mathcal{L}$ is a string $P \in (\mathcal{F} \cup V)^*$, such that $S \overset{*}{\Rightarrow} P$, and $P \neq S$.

Example 5. Recall the DSL from Example 1, where x_0 is the only program input:

$$S \rightarrow select(S, M) \mid transpose(S) \mid x_0$$
$$M \rightarrow 1 \mid 2 \mid 3 \mid ... \mid 10$$

Some examples of sketches include $select(S, M)$, $transpose(S)$, and $select(transpose(S), M)$. Moreover, $select(S, 1)$, $transpose(x_0)$, and $select(transpose(S), 1)$ are not sketches because they contain terminal symbols corresponding to constants and inputs.

3.1. K-tree Model For Sketches

The first step in our enumeration algorithm requires us to encode the space of syntactically correct sketches. Similarly to programs, sketches can be represented as abstract syntax trees. Our idea is to implicitly represent all abstract syntax trees using a set of constraints in SMT, and then use an SMT solver to enumerate them. We achieve this by using the k-tree model for sketches. A k-tree is simply a tree in which all nodes but those at maximum depth have k children. We think of k-tree nodes as structures with 3 attributes, (1) a pointer to a DSL component or *null*, (2) a list of pointers to its children, and (3) a unique index. We use s_N

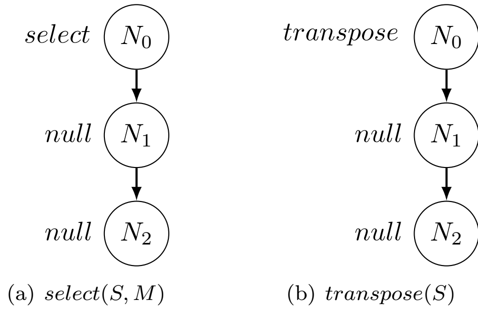
[1] An SMT solver is program that can verify the satisfiability of SMT formulas.

[2] The terminal symbols of the 3 classes are pairwise disjoint, thus $\mathcal{F} \cap \mathcal{I} = \emptyset$, $\mathcal{F} \cap \mathcal{C} = \emptyset$, $\mathcal{I} \cap \mathcal{C} = \emptyset$.

to the unique index associated with node N. If we choose k as the maximum component-arity[3] of all DSL components, then we are guaranteed to be able to represent all sketches of that DSL using k-trees.

Example 6. Consider the DSL from Example 5. The component-arity of $select(S, M)$ is 1, because out of its 2 arguments, only S can be converted into a component. The component-arity of $transpose(S)$ is equal to its arity, because its only input S can be converted into a component.

Example 7. Consider the DSL from Example 5. Since the maximum component-arity of all DSL components is 1, a 1-tree suffices to represent all sketches in this DSL. In Figure 2, we show some examples of sketches of this DSL represented in a 1-tree of depth 2.



(a) $select(S, M)$ (b) $transpose(S)$

Figure 2: Two sketches in the DSL of Example 5 represented in a 1-tree of depth 2.

Next, we focus on our SMT encoding of the k-tree model for sketches.

Variables Our goal is to encode all sketches of a given DSL $\mathcal{L}$ in SMT using integer variables. Therefore, we need to associate each DSL component with a unique positive integer identifier. For each DSL component c, we use $\textsc{Id}(c)$ to denote the unique positive integer identifier associated with it. We also say that $\textsc{Id}(null) = 0$. Let us consider an abstract k-tree T of depth d, where k is the maximum component-arity of components in $\mathcal{L}$. To encode this tree in SMT, we will need two sorts of variables:

- An integer variable v_{s_N} for each node N of the k-tree T, denoting the DSL component assigned to node N;
- An auxiliary Boolean variable b_{s_N} for each node N of the k-tree T, denoting whether node N is assigned to a component ($b_{s_N} = 1$), or to $null$ ($b_{s_N} = 0$).

[3]The component-arity of a given component e is the number of components e can take as input.

Constraints Before discussing the constraints, we introduce some notation that will helps us describe them. Given a k-tree T, we use $\text{Int}(T)$, and $\text{Nodes}(T)$ to refer to the internal, and all nodes of T, respectively. We use $\text{Ch}(N)$ to refer to children of node N. Given a DSL $\mathcal{L}$, we use $\text{Comps}(\mathcal{L})$, and $\text{ComponentStartSymbol}(\mathcal{L})$ to refer to its components, and the components with a production rule whose left-hand side is the start symbol of the DSL, respectively. Lastly, we define the function $\text{CDomain}(C|N = e)$, read the component domain of node $C \in \text{Ch}(N)$ given that Node N has been assigned to component e. $\text{CDomain}(C|N = e)$ is determined by the arguments of e: if node C is the i'th child of N, and component e has a component-arity bigger or equal to i, then $\text{CDomain}(C|N = e)$ is the set DSL components that can be assigned to i'th argument of component e; if node C is the i'th child of N, and component e has a component-arity smaller than i, then $\text{Domain}(C|N = e) = \{null\}$. The constraints are as follows.

- One constraint asserting that the output node must be assigned to components with a production rule whose left-hand side is the start symbol of the DSL:

$$\bigvee_{c \,\in\, \text{ComponentStartSymbol}(\mathcal{L})} v_0 = \textsc{Id}(c) \qquad (1)$$

Example 8. Consider the DSL from Example 5. Constraint 1 on the output node of any 2-tree of this DSL is as follows:

$$v_0 = \textsc{Id}(select) \lor v_0 = \textsc{Id}(transpose)$$

- Constraints ensuring consistency between parent and child nodes (enforcing the structure defined in the grammar):

$$\forall N \in \text{Int}(T), \forall C \in \text{Ch}(N), \forall c_1 \in \text{Comps}(\mathcal{L}) :$$
$$v_{s_N} = \textsc{Id}(c_1) \implies \bigvee_{\substack{c_2 \in \text{CDom}(C|N=c_1) \\ \cup\{null\}}} v_{s_C} = \textsc{Id}(c_2) \qquad (2)$$

Example 9. Consider the DSL from Example 5, and the 1-tree of Figure 2(a). Constraint 2 on the internal node N_1, for component $transpose$ is as follows:

$$v_1 = \textsc{Id}(transpose) \implies (v_2 = \textsc{Id}(select) \lor v_2 = \textsc{Id}(transpose) \lor v_2 = \textsc{Id}(null))$$

- Constraints ensuring that if internal nodes are assigned $null$, then their children are assigned $null$ as well:

$$\forall N \in \text{Int}(T), \forall C \in \text{Ch}(N) : \quad v_{s_N} = \textsc{Id}(null) \implies v_{s_C} = \textsc{Id}(null) \qquad (3)$$

Example 10. Consider the DSL from Example 5, and the 1-tree of Figure 2(a). Constraint 3 on the internal node N_1 is as follows:

$$v_1 = \textsc{Id}(null) \implies v_2 = \textsc{Id}(null)$$

Optional constraints We can further reduce the program space by adding extra constraints.

- Constraints used to enforce exactly p components to be used in the k-tree:

$$\forall N \in \text{Nodes}(T): \\ v_{s_N} = \textsc{Id}(null) \iff b_{s_N} = 0 \tag{4}$$

$$\Big(\sum_{N \in \text{Nodes}(T)} b_{s_N} \Big) = p \tag{5}$$

Example 11. Consider the DSL from Example 5, and the 1-tree of Figure 2(a). If we want to enforce the generated program to have 2 components, we add the following constraints:

$$v_0 = \textsc{Id}(null) \iff b_0 = 0$$
$$v_1 = \textsc{Id}(null) \iff b_1 = 0$$
$$v_2 = \textsc{Id}(null) \iff b_2 = 0$$
$$(b_0 + b_1 + b_2) = 2$$

- One constraint used to enforce at least one the leaf nodes to be used:

$$\bigvee_{N \in \text{Nodes}(T)} \neg(v_{s_N} = \textsc{Id}(null)) \tag{6}$$

Example 12. Consider the DSL from Example 5, and the 1-tree of Figure 2(a). If we want to enforce the leaf to be used, we add the following constraint:

$$\neg(v_2 = \textsc{Id}(null))$$

Given a DSL, we can construct a formula with the all of the previously described constraints, and then use an SMT solver enumerate satisfying assignments of that formula. Each of the satisfying assignments corresponds to a different sketch.

3.2. Sketch Completion

The second step in our enumeration algorithm is the sketch completion. Given a particular sketch, our objective is to exhaustively enumerate the complete programs that can be obtained from that sketch. We can do this step by using a graph-based search algorithm. Our idea is as follows. The structure of the AST of a program is solely decided by its sketch. Therefore, after enumerating a sketch, we can construct an incomplete AST with the correct structure, in which Leaf nodes are not assigned to any DSL element. Example 13 illustrates this data-structure for a particular sketch.

Example 13. Consider the DSL of Example 5 and the sketch $select(transpose(S), M)$. Figure 3 illustrates the incomplete AST of this sketch. Leaf nodes are left unassigned.

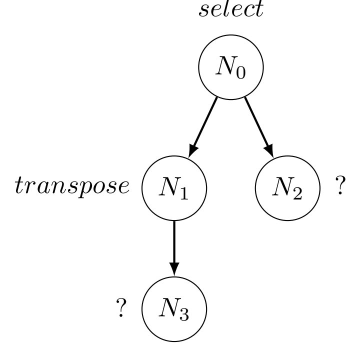


Figure 3: Incomplete AST for the sketch $select(transpose(S), M)$.

Algorithm 1 shows the DynamicFillTree function that leverages this data structure in order to enumerate all programs of a sketch. It resembles a depth-first search on a graph. The result of calling DynamicFillTree on a AST node T is a list with all complete AST with root node T. DynamicFillTree uses two core functions: (1) Combinations, and (2) FillLeaves.

The function Combinations (line 7) returns a list where each element is a combination of sub-trees. For instance, let us suppose that Combinations is called on the list $[[t_1, t_2], [t_3, t_4], [t_5]]$. The possible combinations are $[\{t_1, t_3, t_5\}, \{t_1, t_4, t_5\}, \{t_2, t_3, t_5\}, \{t_2, t_4, t_5\}]$.

The elements assigned to the leaf nodes are decided by the FillLeaves function (line 8). This function returns all possible combinations of assignments to the leaf nodes of a given internal node. In order to prevent the generation of semantically incorrect programs, this function depends on not only of the component assigned to the parent node but also on the sub-trees of its non-leaf siblings.

Example 14. Consider the DSL from Example 5, and the AST from Figure 3. Suppose that the input table x_0 is a table with 3 rows and 6 columns. Calling FillLeaves on the children of node N_0 yields the following: FillLeaves$([N_2], [N_0 = select, N_1 = transpose, N_3 = x_0\}) = [\{N_2 = 1\}, \{N_2 = 2\}, \{N_3 = 3\}]$.

3.3. Synthesis Algorithm

In the classical version of program synthesis, the synthesizer is given a black-box function that either accepts or rejects a program. Thus, if the synthesizer finds a program that is classified as correct

Algorithm 1: DynamicFillTree(T)

```
input  : T, an AST node
output: trees, a list with all possible completions of T
1  child_trees ← []                // a list of lists of complete trees
2  for c ∈ InternalChildren(T) do          // the non-leaf children of T
3      child_trees ← child_trees + [DynamicFillTree(c)]  // all completions starting at c
4  if child_trees = [] then           // if we reach a node whose children are only leaves
5      child_trees ← [[null]]         // we add a dummy value to get a single combination
6  trees ← []
7  for combination ∈ Combinations(child_trees) do
8      constants_list ← FillLeaves(LeafChildren(T), combination ∪ {T = T.v})
9      for constants ∈ constants_list do
10         trees ← trees + [combination ∪ constants ∪ {T = T.v}]
11 return trees
```

by the black-box function, it can simply return that program. In the generalized version, the synthesizer takes as input a cost function it must minimize. In order to consider a program correct (and thus return it), the synthesizer must prove that the cost of the program it has found is minimum. If the synthesizer knows the cost function's minimum value, it can use it as a stopping criterion. However, in general, the synthesizer does not know which kind of cost function the user will provide. One way to solve this problem is by simply enumerating all possible programs of the DSL, calculating their costs, and then select the minimum. In Algorithm 2, we show a generalized synthesis algorithm CostSynthesize that uses this method.

CostSynthesize leverages both the k-tree formula for sketches, and the data structure previously described. We explore program spaces of different depths incrementally (i.e. programs of smaller depth are explored first). In line 3, we start by building the k-tree formula for sketches. The formula should include the optional constraint 6, which enforces at least one of the leaf nodes to be used. This constraint prevents the same sketch from being enumerated twice at different depths. After enumerating a sketch in line 4, we construct the sketch's incomplete AST by calling the function BuildTree in line 6. Subsequently, in the **for** loop of lines $7-10$, we sweep through all possible completions of the incomplete AST, and test them against the problem specification. In line 7, we call the function DynamicFillTree(T), which returns a list with all possible completions of T. In lines $8-11$, we convert each tree into a program in the DSL, and test it against the problem specification.

Although CostSynthesize theoretically solves the generalized synthesis problem, it is a very inefficient algorithm, since it goes through the entire program space. In real-world scenarios, this is simply not a feasible approach. We can tackle this inefficiency problem at the expense of correctness by only exploring the first N ranked programs[4]. Our motivation is that if we explore the program space orderly, we do not have to go through the entire program space to find a program that is likely to be correct. In fact, if we have a sufficiently good program ordering, we might even find the optimal program. The ordering of the program space can be achieved by encoding adequate predicates into the SMT encoding of the k-tree model, which we will discuss next. To implement this non-optimal version of the synthesis algorithm, the only modification we have to make to CostSynthesize is to add a timeout to each iteration of the **for** loop of lines $2-12$. In this way, the algorithm will only explore the highest ranking programs according to the encoded predicates.

3.4. Program Space Ordering

The current SMT encoding of k-tree model provides us a way to perform a brute force enumeration of sketches. However, simply performing a blind brute force search usually does not suffice, due the size of the program space[13]. One way to tackle this problem is to perform an heuristically guided search. We can achieve this by encoding predicates into the SMT formula of the k-tree, each one stating a degree of preference towards a different construct. For example, we could create a predicate $occurs(select, 5)$, stating that we would like component *select* to appear in the synthesized

[4] If we do so, CostSynthesize is only guaranteed to return the best among the explored programs. Therefore, the solutions it provides are not necessarily correct.

Algorithm 2: COSTSYNTHESIZE($\mathcal{L}, c, \Lambda, k, n$)

```
input  : L, a domain specific language
         Λ, a problem specification
         c, a cost function
         k, the maximum component-arity of any component in the DSL
         n, an upper-bound for depth of the solution
output: P_best, a program with depth at most n such that P = argmin_{P'∈L} c(P', Λ)
1  P_best ← ∅
2  for i = 0 to n do
3      F ← BuildSketchFormula(L, k, i)      // build the k-tree formula for sketches
4      ω ← SMTSolve(F)                      // ω is a sketch
5      while ω ≠ ∅ do
6          T ← BuildTree(L, F, ω)           // convert the sketch to an incomplete AST
7          for T_c ∈ DynamicFillTree(T) do  // for every completion of T
8              P ← TreeToProgram(T_c)
9              if c(P, Λ) ≤ c(P_best, Λ) then
10                 P_best ← P
11         F ← BlockModel(F, ω)             // block the current sketch
12         ω ← SMTSolve(F)
13 return P_best
```

program with a preference value of 5. The idea is that if we encode these predicates into the SMT formula of the k-tree, we can ask the SMT solver to enumerate the programs orderly, starting with the ones that maximize the preferences. Essentially, we are transforming the SMT solving problem into a MaxSMT solving problem, where the goal is to find the program with the highest score[2]. Next, we provide a way to encode two different kinds of predicates for DSL components: (1) $occurs(e, w)$ used to state that we would like component e to occur in the program with a preference value of w, and (2) $is_parent(e_1, e_2, w)$ used to state what we would like the sequence of components $e_1(e_2(...), ...)$ to occur in the synthesized program with a preference value of w.

Variables To calculate a program's score we need to know which predicates it satisfies. Thus, for each predicate we have a Boolean variable stating if it is satisfied.

- A Boolean variable o_e for each predicate $occurs(e, w)$, such that $o_e = 1$ if and only if component e occurs;
- A Boolean variable p_{e_1,e_2} for each predicate $is_parent(e_1, e_2, w)$, such that $p_{e_1,e_2} = 1$ if and only if the sequence of components $e_1(e_2(...), ...)$ occurs;
- An auxiliary Boolean variable $p_{e_1,e_2,N}$ for each predicate $is_parent(e_1, e_2, w)$ and internal node N, such that $p_{e_1,e_2,N} = 1$ if and only if the sequence of components $e_1(e_2(...), ...)$ occurs and starts at node N.

Objective Function We use Occurs($\mathcal{L}$), and IP($\mathcal{L}$) to denote the $occurs$, and is_parent predicates the user has provided for the DSL $\mathcal{L}$. The objective function the SMT solver has to maximize is as follows.

$$\sum_{occurs(e,w)\,\in\,\text{Occurs}(\mathcal{L})} w \cdot o_e + \sum_{is_parent(e_1,e_2,w)\,\in\,\text{IP}(\mathcal{L})} w \cdot p_{e_1,e_2} \tag{7}$$

Example 15. Consider the DSL from Example 5, and the following predicates:

$$occurs(select, 10) \tag{8}$$
$$occurs(transpose, 2) \tag{9}$$
$$is_parent(select, transpose, 1) \tag{10}$$
$$is_parent(transpose, transpose, 10) \tag{11}$$

The program $select(transpose(x_0), 1)$ satisfies predicates 8, 9, and 10. Therefore, it has a score of $10 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 + 10 \cdot 0 = 13$.

Constraints We use Int(T), and Nodes(T) to denote the internal nodes, and all nodes of k-tree T, respectively. We also use Ch(N) to denote to children of node N. Id is the function that maps DSL components to their unique positive integer identifiers.

- Constraints asserting that if a predicate in Occurs($\mathcal{L}$) is satisfied, then the corresponding Boolean variable is assigned to 1, or 0 if otherwise.

$$\forall occurs(e,w) \in \text{Occurs}(\mathcal{L}): \bigvee_{N \in \text{Nodes}(T)} v_{s_N} = \text{Id}(e) \iff o_e = 1 \tag{12}$$

Example 16. Consider the DSL from Example 5, the 1-tree of depth 2 from Figure 2(a), and the predicate $occurs(select, 10)$. Constraint 12 for the predicate $occurs(select, 10)$ is as follows:

$$\bigvee_{i=0}^{3} v_i = \text{Id}(select) \iff o_{select} = 1$$

- Constraints asserting that if a predicate in IP($\mathcal{L}$) is satisfied, then the corresponding Boolean variable is assigned to 1, or 0 if otherwise.

$$\begin{aligned} &\forall is_parent(e_1, e_2, w) \in \text{IP}(\mathcal{L}), \forall N \in \text{Int}(T): \\ &v_{s_N} = \text{Id}(e_1) \wedge (\bigvee_{C \in \text{Ch}(N)} v_{s_C} = \text{Id}(e_2)) \\ &\iff p_{e_1,e_2,N} = 1 \end{aligned} \tag{13}$$

$$\forall is_parent(e_1, e_2, w) \in \text{IP}(\mathcal{L}): \bigvee_{N \in \text{Int}(N)} p_{e_1,e_2,N} = 1 \iff p_{e_1,e_2} = 1 \tag{14}$$

Example 17. Consider the DSL from Example 5, the 1-tree of depth 2 from Figure 2(a), and the predicate $occurs(select, 10)$. For the predicate $is_parent(select, transpose, 1)$, constraints 13, and 14 are as follows:

$$\begin{aligned} &(v_0 = \text{Id}(select) \wedge (v_1 = \text{Id}(transpose)) \\ &\iff p_{select,transpose,N_0} = 1 \\ &(v_1 = \text{Id}(select) \wedge (v_2 = \text{Id}(transpose)) \\ &\iff p_{select,transpose,N_1} = 1 \\ &(p_{select,transpose,N_0} \vee p_{select,transpose,N_1}) \\ &\iff p_{select,transpose} \end{aligned}$$

Since these predicates are fully encoded into the SMT formula of the k-tree, we do not need to make any change to the synthesis algorithm. Notwithstanding the usefulness of these predicates to guide the search, we would like to emphasize that there is a cost for each one we add. In particular, each $is_parent(e_1, e_2, N)$ adds an exponential number of variables and constraints. The more of these predicates we add, the more complex the optimization problem will be.

4. Evaluation

4.1. Experimental Setup

In order to evaluate CostSynthesize shown in Algorithm 2, we designed a DSL for table manipulation that closely resembles R. In our DSL every component has a component arity of at most 2. Therefore, we used a 2-tree model to encode the sketch space. Subsequently, we generated 487 benchmarks, each one comprised of: (1) an input table, (2) a noisy output table, (3) and a correct output table. The output tables were comprised of two columns, one of strings, and one of numerical data. The noisy output tables were obtained by introducing error into the numerical columns of the correct output tables. The method used to generate these benchmarks can be found in the thesis. In Table 1, we present a summary of the benchmarks.

Solution Depth	3	4	5	6
# Tests	164	147	101	75

Table 1: Brief summary of the 487 generated benchmarks.

Algorithm 3 shows the cost function we used to score programs. We used a threshold of $\epsilon = 0.10$ (line 18 of Algorithm 3). Moreover, we encoded both is_parent and $occurs$ predicates in the k-tree formula for sketches. Finally, there is one difference between our implementation and the algorithms provided in this paper. Although CostSynthesize is designed to search for solutions of different depths, in our tests, we provided the synthesizer with the depth of the solution we found ourselves, and forced the synthesizer to search for a solution of that depth. This was done because we sought to evaluate the performance of the synthesizer on feasible program spaces.

We ran each benchmark with a time limit of 3600 seconds, on a Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz with 64GB of RAM. We used the Z3[1, 3] SMT solver to enumerate sketches.

4.2. Results and Discussion

We consider having solved a benchmark if the output table produced by the best solution found by the synthesizer within the time limit is equal to the correct output table of the benchmark. In 261 out of 487 benchmarks, a correct solution was enumerated at some stage during the execution. However, we were only able to solve 203 out of the 261 bench-

Algorithm 3: $\textsc{CostFunction}(P, \Lambda)$

```
input  : P, a program
         Λ, a program specification
output: c, the cost of program P on specification Λ
1  c ← 0
2  for (x⃗_i, y_i) ∈ Λ do
3      ŷ_i ← P(x⃗_i)
4      if NRows(ŷ_i) ≠ NRows(y_i) or NCols(ŷ_i) ≠ NCols(y_i) then
5          return +∞  // if tables have different column types (ex int,int vs str,int)
6      categorical_1 ← Sort(CategoricalCol(ŷ_i))        // sorted alphabetically
7      categorical_2 ← Sort(CategoricalCol(y_i))        // sorted alphabetically
8      if categorical_1 ≠ categorical_2 then
9          return +∞  // if categorical columns are different, then program is wrong
10     y_i ← Sort(y_i, key = lambda x : Categorical(x), Numerical(x))
11     ŷ_i ← Sort(ŷ_i, key = lambda x : Categorical(x), Numerical(x))
12     numerical_1 ← NumericalCol(ŷ_i)
13     numerical_2 ← NumericalCol(y_i)
14     for i = 0 to i = NRows(y_i) do
15         d ← |(numerical_1[i] − numerical_2[i]) / (MaxValue(y_i) − MinValue(y_i))|  // normalizing distance to the interval of [0, 1]
16         if d > ε then
17             return +∞  // if there is a deviation of more than ε, discard the program
18         c ← c + d / NRows(y_i)
19 return c
```

marks, that is, the synthesizer enumerated a correct solution but failed to select it in 58 out of the 261 benchmarks (due to cost being lower on incorrect solutions). In Figure 4, we summarize the obtained results by solution depth. We can see that we are able to solve instances with only up to depth 4. The pace at which the synthesizer is enumerating programs is not sufficiently fast to reach a correct solution within the time limit for instances of depth 5. Profiling one of the instances we see that 77% of the execution time is spent on the construction of programs given a sketch (the sketch completion step), 8% is spent on the cost function, 5% is spent on the SMTSolve calls used to enumerate sketches, and the remaining 10% is spent on framework-specific function calls (initialization functions and others). The reason the majority of execution time is spent on the programs' construction is that it requires us to evaluate programs using the DSL's interpreter. Therefore, improving the synthesizer's performance involves either reducing the number of evaluations (namely through the use of pruning techniques), or improving the interpreter's performance.

4.3. Impact of Pruning

In order to reduce the number of evaluations, and thus increase the synthesizer's performance, we decided to implement Morpheus[6] pruning technique into our algorithm. Morpheus uses abstract specifications of components to discard incomplete programs without having to complete them. A further explanation of this pruning technique can be found in the thesis. In Figure 5, we compare the times at which the first correct program was enumerated, with and without pruning. In general, we can that the first correct solution is being enumerated first when pruning is used. There are 43 instances in which we are only able to enumerate a correct solution if we use pruning. However, there are also 4 instances in which we are only able to enumerate a correct solution if we do not use it.

5. Conclusions

In this paper, we proposed a generalized algorithm to solve the problem of synthesizing programs from noisy examples. The key challenge we addressed is the fact that these examples only roughly approximate the desired input-output behavior. Our idea is to enumerate programs orderly within a fixed time

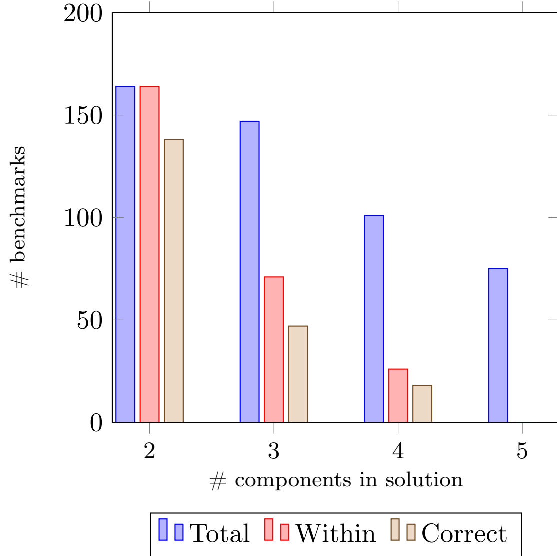


Figure 4: Number of solved benchmarks by depth. For each depth, the blue bar represents the total number of benchmarks with the given depth. The red, and yellow bars represents the number of benchmarks in which a correct program was enumerated, and in which a correct program selected by the synthesizer, respectively.

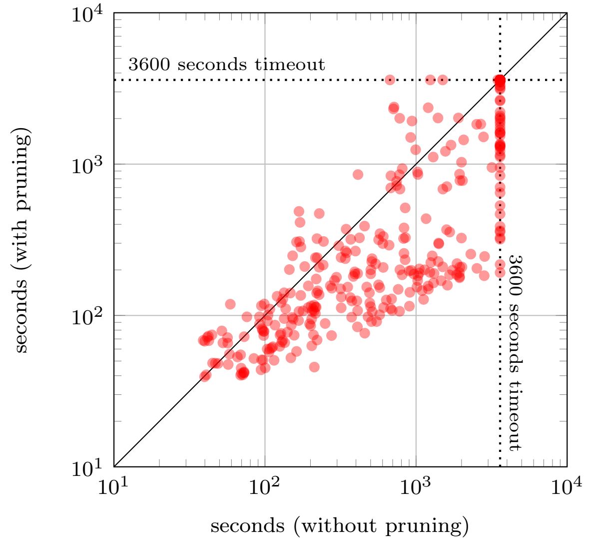


Figure 5: Time at which the first correct solution was enumerated for all benchmarks within the time limit of 3600 seconds.

limit, and then select the best among all using a cost function. We evaluated this approach in a scenario where the problem specification is comprised of an input table, and a noisy output table. We also showed the impact of pruning through the use of incomplete specifications of components. Finally, we built a real program synthesizer and integrated it into the Trinity[16] tool.

5.1. Future Work

There is a lot of promising work being currently developed in the field of program synthesis. In particular, accelerating the enumeration process is still a very active area of research. Recently, **(author?)**[21] proposed a synthesis algorithm that can achieve a throughput of 31,400 programs per second. We intend to explore ways to achieve such a similar or better throughput, since we could see huge improvements in our algorithm from it.

We also intend to explore different ways to disambiguate programs. One of the drawbacks of using incomplete specifications such as input-output examples is that there is usually a big set of programs that satisfy it, albeit most of them are not representative of the user's intent. In our case, we have an extra layer of complexity, since we aim to synthesize programs from noisy input-output examples. One idea to tackle this problem is to interact with the users whilst synthesizing programs, in order to better understand their intent. For example, suppose that the synthesizer finds two programs that both yield the desired output on the user's provided input. In order to select the correct program among the two, the synthesizer could try to search for a different set of inputs in which both programs yield different outputs. Subsequently, the synthesizer could ask the user to select the correct output for the new inputs.

There is also some recent work that makes use of other sources of information to reduce the program space. For example, Draco[17] is a recent program synthesizer for the *Vega-lite*[20] language that requires the user to provide some (but not all) language elements to be used. We intend to explore a similar idea by using predicates, in which we require the user to provide one of the components that is necessarily part of the solution.

Finally, parallelization is still to be fully explored in program synthesis. Our two-step enumeration algorithm provides a straightforward way to introduce parallelization. In our enumeration algorithm, searching for complete programs in different sketches represent independent tasks. Therefore, we can always explore an arbitrary number of sketches at the same time, granted that the generation of sketches outpaces its consumption.

Acknowledgements

This work was supported by national funds through FCT with references UID/CEC/50021/2019, CMU/AIR/0022/2017 and DSAIPA/AI/0044/2018.

References

[1] N. Bjørner, A. Phan, and L. Fleckenstein. νz - an optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 194–199, 2015.

[2] Y. Chen, R. Martins, and Y. Feng. Maximal multi-layer specification synthesis. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.*, pages 602–612, 2019.

[3] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 345–356. ACM, 2016.

[5] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018.

[6] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. 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, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[7] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[8] J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815. ACM, 2016.

[9] A. Gascón, A. Tiwari, B. Carmer, and U. Mathur. Look for the proof to find the program: Decorated-component-based program synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 86–103, 2017.

[10] C. C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 219–240, 1969.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[13] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[14] Z. Jin, M. R. Anderson, M. J. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 683–698, 2017.

[15] V. Le and S. Gulwani. Flashextract: a framework for data extraction by examples. In M. F. P. O'Boyle and K. Pingali, editors, *ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, pages 542–553. ACM, 2014.

[16] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12):1914–1917, 2019.

[17] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Trans. Vis. Comput. Graph.*, 25(1):438–448, 2019.

[18] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

[19] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[20] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, 2017.

[21] K. Shi, J. Steinhardt, and P. Liang. Frangel: component-based synthesis with control structures. *PACMPL*, 3(POPL):73:1–73:29, 2019.

[22] C. Smith and A. Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 326–340, 2016.

[23] A. Tiwari, A. Gascón, and B. Dutertre. Program synthesis using dual interpretation. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 482–497, 2015.

[24] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466, 2017.

[25] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[26] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 224–234, 2013.