Program Synthesis from Noisy Tabular Data

Daniel Rosa Ramos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Vasco Miguel Gomes Nunes Manquinho
Professor Maria Inês Camarate de Campos Lynce de Faria

Examination Committee

October 2019

Acknowledgments

I would like to thank my supervisors, Professor Inês Lynce, and Professor Vasco Manquinho, for their continuous guidance throughout the last year.

I would also like to thank Duarte David, Margarida Ferreira, and Ricardo Brancas who have read the many-versions of this work.

A word of gratitude goes to Rúben Martins for his insightful comments, and for his hospitality.

I would also like to express my sincere gratitude to my family and friends for their unwavering support.

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

Resumo

O foco desta dissertação é o problema de síntese de programas. O interesse na área de síntese de programas tem aumentado nos últimos anos, principalmente devido a uma das suas grandes promessas: permitir que qualquer utilizador consiga construir programas sem ter de saber programar, a partir de exemplos *input-output*. As mais recentes propostas de sintetizadores permitem gerar programas através de exemplos, contudo estes raramente suportam a existência de ruído. Isto significa que uma pequena gralha num exemplo por parte do utilizador pode tornar a tarefa de sintetizar um programa inexequível. Para além disso, sintetizar programas a partir de um conjunto de exemplos com ruído continua por explorar. Neste documento procuramos resolver o problema de síntese de programas neste cenário.

Começamos por apresentar os conceitos fundamentais utilizados por sintetizadores de programas. Apresentamos um sintetizador, TRINITY[29], e formalizamos o modelo que este usa para enumerar programas: o modelo k-*tree*. De seguida, propomos um algoritmo alternativo para enumerar programas baseado na ideia de *sketches* (i.e. programas incompletos). O nosso algoritmo divide a enumeração em dois passos: (1) enumeração de *sketches*, e (2) preenchimento de *sketches*. A enumeração de *sketches* é feita recorrendo ao modelo k-tree, enquanto que o seu preenchimento é feito utilizando avaliações parciais e procura em grafos. Avaliamos empiricamente a abordagem proposta, e mostramos que esta supera o TRINITY no domínio da transformação de tabelas. Apresentamos um novo algoritmo de síntese de programas generalizado, que utiliza o algoritmo de enumeração proposto, para resolver o problema de sintetizar programas a partir de exemplos com ruído. Por fim, construímos um sintetizador que consegue sintetizar programas através de exemplos de tabelas com ruído.

Palavras-chave: Síntese de Programas, Satisfatibilidade Módulo Teorias, Exemplos com Ruído, Transformação de Tabelas

Abstract

This dissertation 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 infeasible. 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 start by introducing the fundamental concepts related to program synthesis. We present an existing synthesizer, TRINITY [29], and formally describe the model it uses to enumerate programs: the k-tree model. Subsequently, we propose a new enumeration algorithm based on the idea of sketches (i.e. incomplete programs). Our algorithm divides the enumeration into two steps: (1) sketch enumeration, and (2) sketch completion. The sketch enumeration step is done by adapting the k-tree model, whereas the sketch completion step is done by combining partial evaluations of programs and graph search. By empirically evaluating our enumeration algorithm, we show that it significantly outperforms TRINITY in the domain of table transformations. We propose a generalized program synthesis algorithm, which leverages our enumeration algorithm, to solve the problem of synthesizing programs from noisy examples. Finally, we build a synthesizer capable of synthesizing programs from noisy examples of tables.

Keywords: Program Synthesis, Satisfiability Modulo Theories, Noisy Examples, Table Transformations

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 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 [22]. There has been significant progress in the past few years in the field of program synthesis [7, 24, 43], but the problem of automatically synthesizing programs has long been studied. In 1969, Green [18] showed how theorem provers could be used to automatically synthesize LISP programs. The method required the user to provide a logical formula completely describing the input-output behavior. Besides this method, other approaches that required a similar set of user-inputs were proposed. However, such methods shared the same shortcoming: they required complete specifications of programs, which can be as hard to write as the program itself. This is why the most recent works focus on synthesizing programs from incomplete specifications [22].

1.1 Contributions

This document gives the reader a small overview of program synthesis. We explore different methods of solving the program synthesis problem and discuss their inherent shortcomings. We give particular focus to the problem of synthesizing programs from input-output examples. Furthermore, we propose a new variant of this problem: synthesizing programs from noisy examples of tables. The key challenge is that the noisy examples are only approximations of the desired input-output relation, and cannot be used as hard constraints to refute or accept programs. The readers might ask themselves why this problem is relevant, and why it has not been studied before. To address such concerns, we can think of real use cases in which this formulation can be useful. For instance, consider we want to synthesize programs from input-output examples of tables, where each output table is obtained from a plot analyzer tool used to scrape data from hand-drawn charts. If the output tables are obtained this way, they will necessarily contain some noise, either due to the drawings themselves or the application that interprets them. In such a setting, we do not want to find a program that satisfies every input-output example, because the examples are only general approximations of what the user intends the program to do. Other similar applications exist. For example, we can think of a scenario where we have a dataset comprised of tables, and a plot obtained by performing transformations on those tables. If we seek to recover the program that transformed the dataset into the plot, we can manually interpret the values of the plot, and then use a program synthesizer to search for the program. Example 1 illustrates a specific scenario.

Example 1. In Figure 1.1(a) we show a table of flights. In Figure 1.1(c) we show a bar chart. The bar chart shows the number of flights that go to LIS from each location (2 from GVA, 1 from BOD, and 1 from BER). In Figure 1.1(b) we show the table obtained from the bar chart of Figure 1.1(c) using WebPlotDigitizer[1]. The table from Figure 1.1(b) is only a rough approximation of the bar chart in Figure 1.1(c).

(a)

Id	From	To
1	GVA	LIS
2	OPO	GVA
3	BOD	LIS
4	BER	LIS
5	GVA	LIS

(b)

From	To
BER	0.99029
BOD	1.00693
GVA	2.01387

(c)

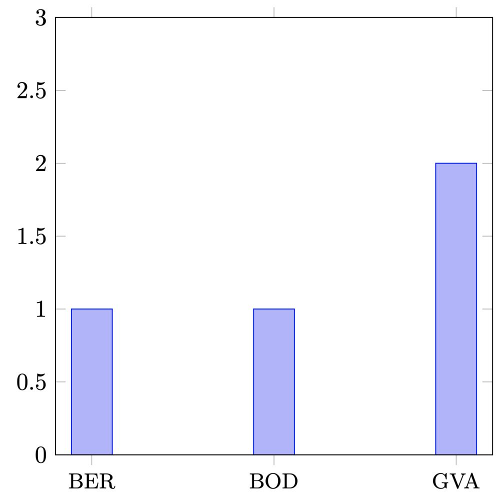


Figure 1.1: Motivating Example.

[1] https://automeris.io/WebPlotDigitizer/

The synthesizer's task is to find a program that maps the table from Figure 1.1(a) to the output table underlying the approximation shown in Figure 1.1(b). In this case, the correct program is shown in Figure 1.2.

```
l1 <- input %>% filter(To == "LIS")
l2 <- l1 %>% group_by(From)
l3 <- l2 %>% summarize(Sum = n())
```

Figure 1.2: Correct program (in R) for the motivating example.

Informally, the statement for our newly proposed problem 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 $P(A) \approx B$? This statement leads us to two questions this dissertation aims to answer:

1. How can we formalize $P(A) \approx B$?
2. Is it possible to leverage existing program synthesis techniques to solve this problem?

Summarizing, this thesis makes the following contributions. We present and discuss a synthesizer that uses the k-tree model, a model used to enumerate programs by program synthesizers. We formalize the SMT encoding of the k-tree model, and discuss its shortcomings by empirically evaluating it on a set of table transformation tasks. We propose a two-step enumeration algorithm based on the idea of sketches that preserves the advantages of the k-tree model, while performing significantly better in the domain of table transformations. We formalize the problem of synthesizing from noisy data, and propose a new program synthesis algorithm that leverages our enumeration algorithm. Finally, we build a synthesizer capable of synthesizing programs from noisy input-output examples of tables.

1.2 Document Structure

This document is organized as follows. In chapter 2, we start with a brief introduction to the program synthesis problem, where we explain the fundamental concepts related to the different dimensions of program synthesizers. In chapter 3, we discuss the different ways of accelerating the synthesis process, namely by the use of deductive reasoning, statistical methods and constraint solving. We also discuss solutions to the most common problems of program synthesizers. In chapter 4, we present a state-of-art synthesizer, TRINITY [29], and explain the model it uses to enumerate programs. We empirically evaluate TRINITY in the domain of table transformations using non-noisy input-output examples. In chapter 5, we propose a new way of enumerating programs that significantly boosts TRINITY performance in the same domain. In chapter 6, we propose a generalized program synthesis algorithm that supports our proposed problem statement. Finally, the thesis concludes in chapter 7.

Chapter 2

Preliminaries

According to Gulwani et al. [6, 22], 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 chapter, 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 capture 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 [17, 18, 20, 49], or ambiguous specifications such as pairs of input-output examples [12, 15, 16, 19, 24, 27, 34, 38, 45, 50, 55] or even natural language [10, 52].

In particular, the focus of this work is 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 [22].

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 a 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[1]. We represent each production rule of a DSL in form $lhs \to rhs$, 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 $\mathcal{A}_0 \to \beta(\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n)$, 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.

We can also associate each component with a specification defining an approximation of its semantics. A relation that maps each component in a DSL to a specification is known as DSL Semantics [14].

Definition 3. (DSL semantics) A finite relation Ψ is a DSL semantics for a DSL $\mathcal{L}$ described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ if it maps each component $\tau \in \Sigma$ to a first-order formula over its n input variables $\vec{x} = [x_0, x_1, x_2, ..., x_{n-1}]$, and its output variable y.

It is important to note that a component's specification does not have to fully specify its semantics.

Example 2. Assuming x_0 is the only program input, we can define a simple DSL $\mathcal{L}$ in the domain of table manipulation:

$$\begin{aligned} S &\to select(S, M) \mid transpose(S) \mid x_0 \\ M &\to 1 \mid 2 \mid 3 \mid ... \mid 10 \end{aligned}$$

We define a DSL Semantics Ψ of $\mathcal{L}$ by mapping each of its components to a first-order formula over its input and output variables:

$$\begin{aligned} \Psi(select) &:= rows(y) = rows(x_0) \land cols(y) < cols(x_0) \\ \Psi(transpose) &:= rows(y) = cols(x_0) \land cols(y) = rows(x_0) \end{aligned}$$

where $rows(x)$ and $cols(x)$ are the number of rows and columns of table x, respectively.

[1]We use "DSL elements" to refer to the components, constants, and inputs of the DSL

Given a DSL, we can define the notion of Program.

Definition 4. (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 5. (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 3. Consider the DSL $\mathcal{L}$ defined in Example 2. Some examples of complete programs are $select(transpose(x_0), 1)$, $select(x_0, 5)$, and $transpose(x_0)$. Moreover, $select(transpose(S), 1)$, $select(S, M)$, and $transpose(S)$ are programs, but they are not complete programs because they contain non-terminal symbols.

Similarly to Feng et al., 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 Root(P). Given a node N, we use Children(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 $R =$ 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. Given a program P, we refer to the leaf nodes of its AST that correspond to non-terminal symbols, input variables, and constants by NonTerminal(P), Variables(P), Constants(P), respectively.

Example 4. Consider the DSL $\mathcal{L}$ defined in Example 2. By successively applying the productions of $\mathcal{L}$ starting from the start symbol, we can derive the program $P := select(transpose(x_0), M)$, which can be represented by the AST illustrated in Figure 2.1.

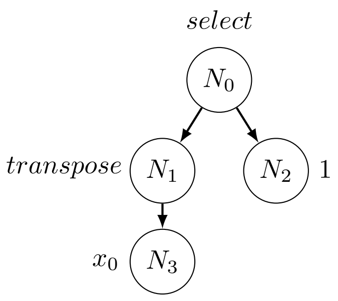


Figure 2.1: AST of the program $select(transpose(x_0), M)$. N_{s_N} denotes the node with index s_N. The DSL symbol associated with each node is next to it.

Using the specification of each component from the DSL semantics we can build a high-level specification for a given program P [14]. The specification of P encodes abstract properties that its output will have given a concrete set of input values.

Definition 6. (Program specification) Given a DSL $\mathcal{L}$, and a DSL Semantics Ψ of $\mathcal{L}$, the program specification of a program $P \in \mathcal{L}$ is a first-order formula Φ_P over its n input variables $\vec{x} = [x_0, x_1, ..., x_{n-1}]$, its output variable y, and k intermediate variables.

$$\Phi_P := \Phi_R \wedge v_{s_R} = y \qquad \text{where } R := \text{Root}(P)$$
$$\Phi_N := \Gamma_N \wedge \bigwedge\nolimits_{N^j \in \text{Children}(N)} \Phi_{N^j}$$

$$\Gamma_N := \begin{cases} \top & \text{if } N \in \text{NonTerminal}(P) \cup \text{Constants}(P) \\ \Omega_N[v_{s_N}/y] & \text{if } N \in \text{Variables}(P) \\ \Omega_N[c_N/\vec{x}, v_{s_N}/y] & \text{otherwise} \end{cases}$$

$$\Omega_N := \begin{cases} y = x_i & \text{if } N \text{ represents input variable } x_i \\ \Psi(\tau) & \text{if } N \text{ represents component } \tau \end{cases}$$

$$c_N := [v_{s_{N^0}}, v_{s_{N^1}}, ..., v_{s_{N^{n-1}}}] \qquad \text{where } [N^0, N^1, ..., N^{n-1}] = \text{ Children(N)}$$

Example 5. Consider the program P defined in Example 4. The program specification Φ_P of P is:

$$\Phi_P := \Gamma_{N_0} \wedge \Gamma_{N_1} \wedge \Gamma_{N_2} \wedge \Gamma_{N_3} \wedge v_0 = y$$
$$\Gamma_{N_0} := rows(v_0) = rows(v_1) \wedge cols(v_0) < cols(v_1)$$
$$\Gamma_{N_1} := rows(v_1) = cols(v_3) \wedge cols(v_1) = rows(v_3)$$
$$\Gamma_{N_2} := \top$$
$$\Gamma_{N_3} := v_3 = x_0$$

The program specification Φ_P states that the number of rows of the output table, $rows(y)$, will be equal to the number of columns of the input table, $cols(x_0)$, since we have $y = v_0$, $rows(v_0) = rows(v_1)$, $rows(v_1) = cols(v_3)$, and $v_3 = x_0$. It also states that the number of columns of the output table, $cols(y)$, will be less than the number of rows of the input table, $rows(x_0)$, since we have $y = v_0$, $cols(v_0) < cols(v_1)$, $cols(v_1) = rows(v_3)$, and $v_3 = x_0$.

We represent the evaluation of a complete program P on input $\vec{x}$, where $\vec{x}$ is a vector of values corresponding to each of the input variables of P, by $P(\vec{x})$.

2.3 Search Technique

Given a DSL, the program synthesizer will have to find a complete program in the DSL that satisfies a given problem specification.

Problem statement 1. Given a domain specific language $\mathcal{L}$, and a problem specification Λ with n input-output examples $(\vec{x}_i, y_i) \in \Lambda$, the program synthesis problem is to find a complete program $P \in \mathcal{L}$ such that $\bigwedge_{i=0}^{n-1} P(\vec{x}_i) = y_i$ is a valid formula. We say that P satisfies the problem specification Λ.

2.3.1 Enumerative Search

Among the techniques used to search by program synthesizers, enumerative search is likely to be the simplest. In enumerative search, the idea is to iteratively enumerate programs until one that satisfies the problem specification is found. The way we perform the enumerative search largely depends on the internal representation we choose. In this section, we focus on top-down enumerative search [22] using graph representations. Top-down enumerative search starts from the start symbol of the DSL and uses the production rules to expand it.

Enumerative search is essentially solving a search problem. To solve a search problem, first we will have to decide on which symbolic structure to use and the adequate operators. For instance, we can represent the program space[2] implicitly by using a state-space formulation, where each state represents a program. The start state represents the program containing only the DSL's start symbol. The goal states are the states that represent complete programs that solve the program synthesis problem. For each state, there is a set of actions corresponding to the production rules of the left-most non-terminal symbol of the program it represents. Applying an action to a state corresponds to applying a production rule to the left-most non-terminal symbol of the program it represents.

An alternative way to represent the program space implicitly is by using an AND-OR graph [36], where the nodes are either terminal or non-terminal symbols of the DSL. A non-terminal symbol always corresponds to an OR node. The children of an OR node are the possible expansions of the non-terminal symbol it represents. A terminal symbol can either be a leaf (if it is a constant, or a variable) or an AND node (if it corresponds to a component). The children of an AND node are the arguments of the component it represents. The initial node is an OR node that corresponds to the start symbol of the DSL. In this case, a solution is a sub-graph of the underlying AND-OR graph that represents a complete program that solves the program synthesis problem.

The AND-OR graph formulation is generally a more concise representation of the program space, since the nodes are shared among programs.

Example 6. Figure 2.2(a) illustrates the path that leads to the sate corresponding to the program of Example 4, in the state-space formulation. Figure 2.2(b) illustrates the sub-graph (of the underlying AND-OR graph) corresponding to the program of Example 4, in the AND-OR graph formulation. In the AND-OR

[2]The program space is the space of all complete programs.

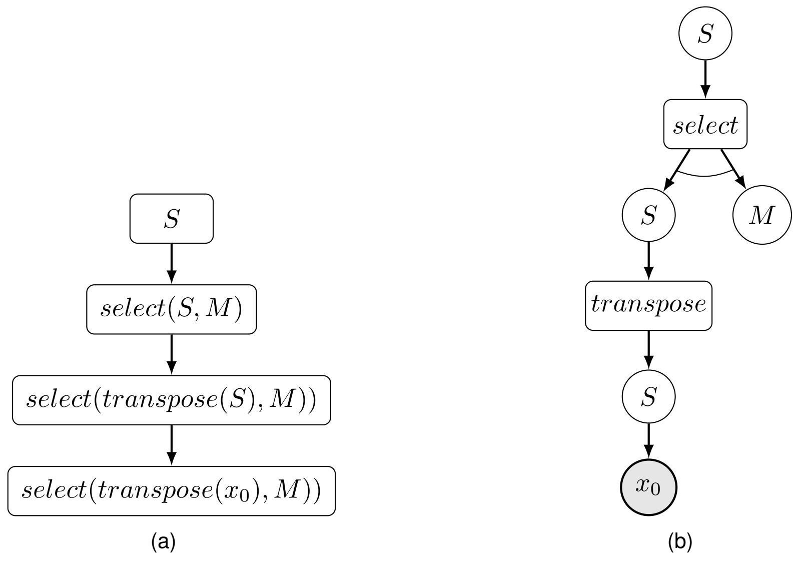


Figure 2.2: Illustrations of the program of Example 4 in the proposed graph based representations.

graph, the OR nodes are represented as white circles, the AND nodes are represented as rectangles, and the LEAF nodes are represented as grey circles.

Since we are solving a search problem, regardless of the model we use, we will always have to face at least two challenges: (1) search space pruning and (2) state ordering.

2.3.2 Search Space Pruning

Before discussing search pruning techniques, 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 φ $\mathcal{T}$-satisfiable if there exists an assignment to each variable of φ such that φ is satisfied.

Example 7. The program specification Φ_P given in Example 5 is a $\mathcal{T}$-formula, where $\mathcal{T}$ is the combination of the theory of Linear Integer Arithmetic (LIA) and the theory of Equality with Uninterpreted Functions (EUF).

When searching for a program that satisfies a problem specification, we can come across incomplete programs that cannot be completed in a way that satisfies it. The objective of search pruning techniques is to find these spurious programs (as soon as possible), in order to prevent them from being further explored. Next, we illustrate how to achieve this using an SMT-based pruning technique.

SMT-based pruning [12] Given a program P, and a problem specification Λ, our goal is to decide if there is a completion of P that satisfies Λ. Recall that the program specification Φ_P encodes abstract properties that the output of P will have given a concrete set of input values. Thus, if we assign the input variables of P to a concrete set of input values from an input-output example $\lambda \in \Lambda$, we can test whether the properties of λ's output are consistent with those obtained from formula Φ_P. We can achieve this by checking the satisfiability of each $\mathcal{T}$-formula $\Phi_P \wedge \Phi_\lambda$, where Φ_P is the program specification of P, and Φ_λ is a $\mathcal{T}$-formula that encodes the properties of example $\lambda \in \Lambda$.

Example 8. Consider a problem specification Λ consisting of one input-output example (x_0, y). The input x_0 is the table represented in Table 2.1, and the output y is the table represented in Table 2.2.

Table 2.1: Input table of Example 7.

1	2	3
4	5	6
7	8	9

Table 2.2: Output table of Example 7.

1	4	7
2	5	8
3	6	9

The $\mathcal{T}$-formula Φ_0 encodes the properties of this input-output example, where $\mathcal{T}$ is the combination of the theory of linear integer arithmetic (LIA) and the theory of equality with uninterpreted functions (EUF).

$$\Phi_0 := rows(x_0) = 3 \wedge cols(x_0) = 3 \wedge rows(y) = 3 \wedge cols(y) = 3$$

Example 9. Consider the program P from Example 4 and the respective program specification Φ_P from Example 5. Consider the problem specification Λ and its abstract specification Φ_0 from Example 8. We test whether P is a suitable program for problem specification Λ by checking if $\Phi_P \wedge \Phi_0$ is $\mathcal{T}$-satisfiabile, where $\mathcal{T}$ is the combination of the theory of linear integer arithmetic (LIA) and the theory of equality with uninterpreted functions (EUF):

$$\begin{aligned}\Phi_P \wedge \Phi_0 := \; & rows(v_0) = rows(v_1) \;\wedge\; \underline{cols(v_0) < cols(v_1)} \wedge \\ & rows(v_1) = cols(v_3) \;\wedge\; \underline{cols(v_1) = rows(v_3)} \wedge \\ & \underline{v_3 = x_0} \;\wedge\; \underline{v_0 = y} \;\wedge\; \underline{rows(x_0) = 3} \wedge \\ & cols(x_0) = 3 \;\wedge\; rows(y) = 3 \;\wedge\; \underline{cols(y) = 3}\end{aligned}$$

Since the formula is not $\mathcal{T}$-satisfiable (we have $cols(y) = 3$, $rows(x_0) = 3$, and $cols(y) < rows(x_0)$), we can discard P, because there is no completion of P that satisfies the example.

The procedure just described and illustrated is sound, but generally not complete [12]. This is due to the program specification being only an approximation of the program's semantics.

2.3.3 State Ordering

If the search is performed blindly, due to the size of the search space, it will probably consume a large number of resources before producing results. Augmenting search algorithms with heuristics is a very common technique to soften this problem. Essentially, heuristics are functions used to guide search procedures. In particular, they can be used to decide which state to expand from a set of states. For example, MORPHEUS [12], a program synthesizer for loop-free programs in R, uses a 2-gram language model to select which incomplete program to choose from the set of incomplete programs being considered. The idea is to select incomplete programs that contain snippets of code that are commonly found together.

Chapter 3

Related Work

Program synthesis is a very wide field, with many different special-purpose algorithms targeting a variety of different applications, such as data preprocessing [3, 12, 19, 27, 21], code repair [44], among others [22]. Synthesizers face different challenges depending on their target domains. For instance, FLASHFILL [19], a programming-by-example synthesizer for the domain of string manipulation, is one of the most prominent program synthesizers. Since it usually works with a very small set of input-output examples (typically 1 or 2), its biggest problem is not to find a program that satisfies the specification[1], but to find the program that the user intends to be synthesized.

In this chapter, we explore different ways of formulating and solving the program synthesis problem. We also discuss the most common problems of program synthesizers and the ideas used to tackle these problems.

3.1 Problem Specifications

Chapter 2 introduced the concept of problem specification. In particular, we defined the problem specification to be a set of input-output examples. However, this does not have to be always the case. There are two types of problem specifications, (1) complete specifications, and (2) incomplete specifications [22].

3.1.1 Complete Specifications

Complete specifications are unambiguous forms of specifying intent. A synthesizer that receives as input a complete specification only returns correct programs. In other words, it is impossible for a program to satisfy a complete specification and not to be correct to the user. Complete specifications must fully cover the input space of the program to be synthesized. Thus, complete specifications are usually specified by means of mechanisms such as first-order logic [22].

[1]Because there is usually a big set of programs that satisfy the specification.

3.1.2 Incomplete Specifications

Incomplete specifications are ambiguous forms of specifying intent. Whenever a specification is incomplete, it is possible for a program to satisfy it while incorrect to the user. A synthesizer that works with incomplete specifications will have to choose which program to return from a set of different programs that satisfy it. This problem is known as the ranking problem. Next, we present two different methods to solve it.

Ranking Functions can be used to solve the ambiguity created by incomplete specifications [38]. Essentially, a ranking function associates a score with each program, thereby creating a preference relation between them. For example, a ranking function following the Occam's razor principle should always prefer simpler programs over complex programs [12]. The idea is to synthesize a set of programs that satisfy the incomplete specification, and then use the ranking function to select the highest scoring program. Synthesizers that use this technique usually must represent a large set of programs concisely. The Version Space Algebra [38, 51, 26] is a data structure commonly used for this purpose. A Version Space Algebra represents a set of programs, and it (potentially) allows for the representation of an exponential set of programs using polynomial space [22]. It resembles the AND-OR graph formulation given in chapter 2. Version Space Algebras also have defined a set of operations such as intersection. For example, the intersection between two Version Space Algebra $V_1 \cap V_2$ is a Version Space Algebra that represents the programs present in both V_1 and V_2.

Disjunctive Program Synthesis [30] is another approach used to solve the ranking problem. Instead of returning a single program, disjunctive program synthesizers return a set of programs, and a mechanism for output selection. Essentially, they postpone the ranking decision to execution time. Given a new input (at execution time), the mechanism runs all programs on the new input, and subsequently selects which program output to return to the user. It decides which output to return by using a set of feature calculators. The feature calculators are functions used to extract features from the outputs.

For example, suppose we want to synthesize a program to remove every negative element from a list. The problem specification we provide to the synthesizer is a single input-output example, the input list $[-3, -1,\ 1,\ 3]$, and the output list $[1,\ 3]$. From the output list, the synthesizer extracts the information that it is a non-empty list, using the feature calculator $NonEmpty(l)$. After finishing the synthesis process, the synthesizer returns: (1) a mechanism for output selection, stating that it prefers outputs that are non-empty lists; and (2) two programs that satisfy the specification, $filter(l, x \geq 0)$, and $filter(l, x \geq 1)$, where $filter(l, p)$ is a function that filters the elements of list l according to predicate p. At execution time, if we provide an input list with only zeros, the mechanism will return the output of the program $filter(l, x \geq 0)$, because it is the only program that returns a non-empty list.

3.2 Program Space

The way we define the program space depends on what we are trying to achieve. Domain specific languages are widely used by program synthesizers whose objective is to synthesize a program from scratch. The synthesizers that solve the program synthesize problem using domain specific languages are said to be solving Syntax Guided Synthesis (SyGuS[2]) problem. The term syntax-guided emerges from the fact that, although the program will be synthesized in an executable programming language, the synthesizer does not have access to its full set of operations. It is limited to the scope of the specified domain specific language. The formal statement of the SyGuS problem is as follows [1]. Given a background theory $\mathcal{T}$, a typed function f, a formula φ over the vocabulary of $\mathcal{T}$ and function f, a set of expressions $\mathcal{L}$ of type of f over the vocabulary of $\mathcal{T}$, the objective is to find an expression $e \in \mathcal{L}$ such that $\varphi[f/e]$ is valid modulo $\mathcal{T}$.

Another way of defining the program space is through the use of sketches. Sketches are incomplete programs with holes (i.e. incomplete parts) that are provided to synthesizer along with the problem specification. For each hole in the sketch, there is a set of possible expressions that can be used to fill it. The task of the synthesizer is to fill each hole, in such a way that the problem specification is satisfied. This technique is known as Sketching [46]. Since the program space is significantly more reduced when good sketches are provided, sketching can be a preferable way of defining the program space when the user has clues about the structure of the program to be synthesized.

3.3 Search Space Pruning

As discussed in Chapter 2, search space pruning is one of the ways of accelerating the enumerative search process. Synthesizers usually prune the search space by using three different deductive based methods, (1) type checking, (2) inverse semantics, and (3) learning. Although pruning techniques do not have to keep all potential solutions, the methods described in this section only perform safe pruning, that is, they never discard parts of the search space that could potentially lead to solutions.

3.3.1 Type-checking

We can divide domain specific languages into two subsets, (1) domain specific languages built for the manipulation of a single data type, and (2) loosely constrained domain specific languages built for the manipulation of different data types. Type-checking is used by synthesizers that work over loosely constrained domain specific languages [22]. It is particularly useful for synthesizers that target algebraic data type (such as lists) manipulation. The reason is that there can be algebraic data types of any type (e.g. lists of integers, lists of strings, lists of lists, and so forth). For example, consider a problem specification with of a set of input-output examples of type $list[\tau_1] \rightarrow list[\tau_2],\ \tau_1 = \tau_2$. In this case, the synthesizer must find a program that takes as input a $list$ of type τ_1 and outputs a $list$ of the same type. This information can be used to discard any program that does not preserve this property. λ^2

[2]SyGuS is also the name of a program synthesis competition held every year.

[15] is a program synthesizer for list manipulation that leverages this idea. It uses enumerative search and it always generates hypotheses (incomplete programs) in a type-aware manner, in order to prevent spurious programs from being explored.

Besides the use of type-checking for search space pruning, there are other approaches that treat the program synthesis problem wholly as a type-checking problem [16, 34, 37].

3.3.2 Inverse Semantics

Given an incomplete program P, we would like to have a mechanism to decide if there is a completion of P that satisfies the problem specification. One way to achieve this is through the use of inverse semantics [22]. This corresponds to asking the question: given a program of form $\mathcal{X}(\phi_1, \phi_2, ..., \phi_k)$, what are the possible expressions that $\phi_1, \phi_2, ..., \phi_k$ can take in order to produce the desired output. For instance, consider a program $select(\phi_1, \phi_2)$, which selects from table ϕ_1 columns according to predicate ϕ_2. If the output we desire from $select(\phi_1, \phi_2)$ is a table with n columns, then we can infer that ϕ_1 must be a table with at least n columns.

MORPHEUS [12] is a tool for automating data wrangling tasks in R that uses this mechanism. MORPHEUS introduced the SMT-based pruning technique explained in section 2.3.2. Given a candidate program P, it uses the DSL semantics to generate a program specification Φ_P, which is subsequently used to detect inconsistencies with the input-output examples. The inconsistencies are detected by using an off-the-shelf SMT-solver such as Z3 [9]. Example 9 illustrates MORPHEUS's search pruning technique. MORPHEUS also uses partial evaluations to further reduce the search space. A partial evaluation consists in evaluating the parts of the program that have been completed, and replacing them by constant values. After the performing partial evaluation of a program P, MORPHEUS performs the SMT-based pruning on top of a new program P_n, where the evaluated parts are replaced by constant values.

λ^2 [15] also uses the idea of inverse semantics to prune the search space. In particular, it pushes the input-output examples downward in order to verify the feasibility of programs. For example, consider a problem specification with a single input-output example comprised of two lists (x_{in}, x_{out}), where $x_{in} = [1, 2, 1]$ and $x_{out} = [2, 3, 4]$. Suppose λ^2 decided to explore a candidate program of form $map(x_{in}, \phi)$, which maps the elements of x_{in} using function ϕ. After considering this candidate program, λ^2 propagates the examples downward to function ϕ. In particular, it generates the input-output examples $\{(1, 2), (2, 3), (1, 4)\}$ as function's ϕ problem specification. Since ϕ is a function, and it is impossible for a function to output different values on the same input, λ^2 detects the inconsistency $\underline{\phi(1) = 2} \wedge \underline{\phi(1) = 4}$ and discards the candidate.

3.3.3 Learning

In the enumerative search process, programs are progressively enumerated until one that satisfies the problem specification is found. Throughout the enumeration process, it is only natural for the programs synthesizers to find multiple spurious programs that do not satisfy the problem specification. We can use the spurious programs found throughout the search to infer similar programs that suffer from the same core problem. The key to this idea is to have a mechanism that can generalize the identified problems.

NEO's [14] framework is based on this idea. For example, consider a problem specification consisting of one input-output example comprised of two lists (x_{in}, y_{out}), where $x_{in} = [1, 10]$ and $y_{out} = [1, 10, 100]$. Suppose NEO decided to explore an incomplete program of $filter(x_{in}, \phi)$, where $filter$ is a function that filters list x_{in} using predicate ϕ. Before starting to exhaustively search for concrete predicates ϕ that might satisfy the problem specification, NEO tests the incomplete program's consistency with the input x_{in} and output y_{out}. Specifically, it uses the method described in section 2.3.2 build a formula that relates the programs' inputs and outputs: $size(y_{out}) \leq size(x_{in})$. Testing this formula against the input-output abstract specification, it concludes that the program is wrong, since the formula is unsatisfiable: $\underline{size(y_{out}) \leq size(x_{in})} \wedge \underline{size(y_{out}) = 3} \wedge \underline{size(x_{in}) = 2}$. Moreover, after exploring this spurious program, NEO infers other programs that suffer from the same core problem: programs whose program specification implies $size(y_{out}) \leq size(x_{in})$. For instance, the specification of $sort(x_{in})$ is $size(y_{out}) = size(x_{in})$. Since $size(y_{out}) = size(x_{in}) \implies size(y_{out}) \leq size(x_{in})$, NEO can infer that $sort(x_{in})$ is also a spurious program.

3.4 State Ordering

3.4.1 Language Models

In natural language processing, it is very common to associate probabilities with sentences. Models that associate probabilities with sentences are known as languages models [25]. Language models have several applications, in particular, they can be used to select the most likely sentence of a set. For instance, an accurate language model for English will associate a higher probability with the sentence *"You're welcome."* than with the sentence *"Your welcome."*. Similarly, we can also think of programs as sentences in a programming language. Just like sentences in a natural language, programs in a programming language are not uniformly distributed [28], that is, they are not all equally likely. Thus, if we build a language model for a programming language, we can then use it to bias the search process, in order to enumerate the most likely programs first.

There are different ways of building languages models. In particular, we can build language models by using probabilistic context-free grammars. A probabilistic context-free grammar is simply a context-free grammar where each production rule $\alpha \rightarrow \beta$ is associated with a conditional probability $Pr(\alpha \rightarrow \beta|\alpha)$ [25]. If we describe a DSL $\mathcal{L}$ through a probabilistic context-free grammar, we can define the probability of a complete program $P \in \mathcal{L}$ to be the probability of selecting each of n production rules $\alpha_i \rightarrow \beta_i$ that lead to it[3]:

$$Pr(P) = \prod_{i=0}^{n-1} Pr(\alpha_i \rightarrow \beta_i|\alpha_i)$$

Example 10. Consider the probabilistic context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where each production rule $r \in R$ is represented by $\alpha \rightarrow \beta \;\; [p]$, where p is the conditional probability $Pr(\alpha \rightarrow \beta|\alpha)$:

[3]Assuming the grammar is unambiguous.

$$
\begin{aligned}
&S \rightarrow select(S, N) && [0.1] \\
&S \rightarrow x_0 && [0.9] \\
&N \rightarrow 1 && [1.0]
\end{aligned}
$$

The probability of the complete program $select(x_0, 1)$ is $0.1 \times 0.9 \times 1 = 0.09$. Furthermore, the sum of the probabilities of all complete programs is $\sum_{n=0}^{\infty} 0.1^n \times 0.9 = 1$.

More generally, Lee et al. define the concept of statistical program model. A statistical program model is a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where each production rule $r \in R$ is associated with a set of k conditional probabilities $Pr(r \mid c^0),\ Pr(r \mid c^1),\ ...,\ Pr(r \mid c^{k-1})$ corresponding to contexts $c^0, c^1, ..., c^{k-1}$. A probabilistic context-free grammar is simply a statistical program model where each production rule is associated with a single conditional probability. In the same way we define the probability of a program using a probabilistic context-free grammar, we can also do it using a statistical program model. Given a complete program P, we use the n production rules that derive it, $r_0, r_1, ..., r_{n-1}$, and the respective contexts, $c_0, c_1, ...c_{n-1}$, to define its probability:

$$Pr(P) = \prod_{i=0}^{n-1} Pr(r_i \mid c_i)$$

Recall that the purpose of using language models is to perform an ordered enumeration of programs. In order words, the objective is to enumerate *likely* programs before *unlikely* programs. We can achieve this by performing search in a weighted graph [28]. Before going into details, note that instead of finding a program with the maximum probability, we can equivalently find a program with the minimum negative log-probability:

$$-log(Pr(P)) = -\sum_{i=0}^{n-1} log(Pr(r_i \mid c_i))$$

Consider the following state-space formulation: states represent programs; goal states correspond to complete programs that satisfy the problem specification; the start state corresponds to the program with the start symbol of the DSL; the actions available at each state are the production rules of the leftmost non-terminal symbol of the corresponding program; the cost of each action is the negative log-probability of selecting the corresponding production rule in the state's context $-log(Pr(r_i \mid c_i))$. The cost of the path from the start state to a goal state that represents program P corresponds to the negative log-probability of P. Therefore, if we use a uniform cost search algorithm (such as Dijkstra's algorithm) on this formulation, we are sure to enumerate the most *likely* programs first. In practice, to speed-up the search, we can also derive a heuristic and use A* [28].

3.4.2 Statistical Program Models

Statistical program models are sufficiently general to cover a wide variety of models [28] used by different synthesizer, namely probabilistic context-free grammars [31], probabilistic high-order grammars [4], n-grams [12, 40], and some neural network models [2, 7]. Next, we discuss one the most prominent statistical program models: DEEPCODER [2].

DEEPCODER's idea is to guide the search by using a neural network. Besides the DSL, DEEPCODER's framework requires (1) a mapping from programs to attributes, and (2) a data generation procedure capable of generating triplets of form $(P, a, \mathcal{E})$, where P is a program, a is vector of attributes that characterize P, and $\mathcal{E}$ is a set of input-output examples of P. The triplets are used to train a neural network to predict a distribution $Pr(a|\mathcal{E})$, where a is a vector of attributes, and $\mathcal{E}$ is a set of input-output examples. The attributes can be arbitrary, in particular, the authors choose to use the presence of components as attributes. For example, consider a DSL that contains the basic arithmetic operators $+, -, \times$. We associate with each program P a vector of attributes a_P, containing a binary entry for each component (the first, second, and third entries denote the presence of $+, -, \times$ in the program, respectively). The attribute vector of program $\times(+(x_0, x_0), x_1)$ is $[1, 0, 1]$. Generating $\mathcal{E}$ by running P over random inputs, we are able to train a neural network $Pr(a|\mathcal{E}, \theta)$ to guess what components the original program contained. After training the neural network with multiple examples, it is possible to query it about which components are likely to appear in a program given the input-output examples. This information can, in turn, be used to sort the productions of rules of the DSL, in order to perform a depth-first search that prioritizes to explore the most likely components first.

Besides DEEPCODER there are similar approaches that rely on neural networks to build a language model [7, 35, 53]. For instance, Parisotto et al. [35] use a neural network that takes as input an incomplete program and the input-output examples, and returns a distribution over the expansions that can be made to the incomplete program. A valid expansion is comprised of the non-terminal symbol to be expanded, and a production rule to be used.

3.5 Counterexample Guided Inductive Synthesis (CEGIS)

Enumerative search is not the only way of solving the program synthesize problem. In particular, it is possible to translate the problem to other formalisms such as SMT, and solve it by using off-the-shelf solvers. Next, we present a framework for solving the program synthesize problem using constraint based techniques, and illustrate it with a concrete example.

In general, the task of finding a program p that satisfies a specification $\phi_{spec}(x, y)$[4] can be modelled in second-order logic [22]:

$$\exists p \forall x, y : \; p(x) = y \implies \phi_{spec}(x, y)$$

However, solving a second-order logic formula is generally infeasible. One idea to workaround this

[4] $\phi_{spec}(x, y)$ is a predicate that is true if and only if (x, y) is a valid input-output pair.

problem is to divide the synthesis process into two steps: (1) searching for a program, and (2) verifying the compliance of the program with the problem specification. In particular, verifying if there exists a valid input-output pair (x, y) that program p does not satisfy can be modelled using first-order logic:

$$\exists x, y : \; p(x) \neq y \wedge \phi_{spec}(x, y)$$

CEGIS [46] is a program synthesis technique that leverages this idea by solving the program synthesis problem in an iterative refinement process. The idea is to synthesize a correct program for a finite set of concrete input-output observations (inductive synthesis), and subsequently verify if it is sufficiently general for the base case. If it does not satisfy the specification for every input, a counter-example is added to the set of concrete input-output observations and the process is repeated. Figure 3.1 illustrates this process.

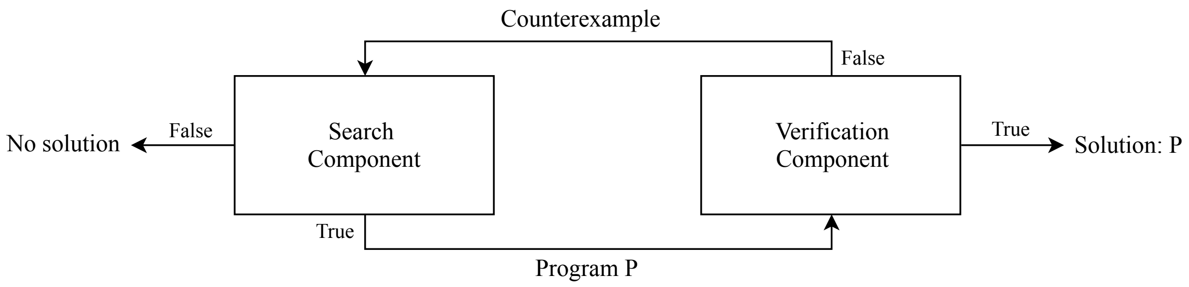


Figure 3.1: CEGIS Diagram.

The intuition behind this method is that by focusing on synthesizing a program for a set sufficiently representative of the set of inputs, the synthesizer will converge faster to the desired program. Next, we present a synthesizer that follows this approach: Brahma [19, 23].

Brahma [19, 23] is a program synthesizer for the domain of bit vectors. BRAHMA requires the user to provide a complete specification using first-order logic formulas:

- A relation $\phi_{spec}(\vec{x}, y)$ on input variables $\vec{x}$ and output variable y, such that $\forall \vec{x}, y \; \phi_{spec}(\vec{x}, y)$ is true if and only if y is the correct output of the program for input $\vec{x}$.

- For each component f_i, a relation $\phi_i(\vec{x}, y)$ on input variables $\vec{x}$ and output variable y, such that $\forall \vec{x}, y \; \phi_i(\vec{x}, y)$ is true if and only if y is the correct output of component f_i for input $\vec{x}$.

It solves the program synthesis problem using the CEGIS approach. Figure 3.1 illustrates the method. At each step, it generates an SMT formula ω_1 whose solution encodes a program that satisfies a set of input-output examples. After finding this solution, it generates another SMT formula ω_2 whose solution is a counterexample to the program generated. If ω_2 is unsatisfiable, it means that no counterexample exists. Therefore, the encoded program satisfies the specification $\phi_{spec}(\vec{x}, y)$.

3.6 Handling Incorrect Examples

In general, the standard formulation of programming by example is to synthesize a program that satisfies all input-output examples. However, most real world applications are prone to errors. It is naive to assume that human-provided input will always be completely correct. A simple mistake, such as mistyping a number in a single input-output example, might go unnoticed by the user, thereby making the problem specification incorrect. Since the synthesizer must satisfy all examples, if specification itself is incorrect, the synthesized program (if it exists) will also be incorrect. In this section, we discuss two methods to handle errors in input-output examples.

The key difference between this research topic and our proposal of synthesizing from noisy examples is that the techniques we discuss next are designed to work on datasets that contain some incorrect examples. In contrast, we aim to solve the problem of synthesizing programs from intput-output examples that only approximate the desired input-output relation. Therefore, the problem specifications we seek to support do not have contain any "correct" example.

3.6.1 Threshold Method

We can interpret the program synthesis problem as a minimization problem, where the objective is to find a program whose cost is below a certain threshold. Generally, the cost of programs is either 1 meaning accept, or 0 meaning reject. However, instead of using this cost function, we can use cost functions that score programs according to their fitness to the examples. We can then select a threshold from which we consider a program correct. For example, Raychev et al. [41] propose to use the number of unsatisfied examples, and the error-rate of the program (the number of unsatisfied examples divided by the number of total examples) as the cost function. Since it might preferable not to satisfy all examples, Raychev et al. also propose to add a regularization term to the cost function. The regularization term penalizes overly complex programs.

3.6.2 Neural Networks

Another idea is to solve the program synthesis problem wholly using neural networks. The intuition is that the statistical nature of neural networks make them less susceptible to errors, which allows them to perceive user intention even when some of the provided examples are wrong. RobustFill [11] uses this idea by training a neural network on randomly generated pairs of input-output examples and programs. The examples are processed at character level, encoded using `ASCII` tokens. The output layer consists of the program tokens (DSL's terminal symbols). The network generates programs token-by-token.

It is also possible to use a neural network to learn the input-output relation, without the need for a DSL. This technique is known as program induction [11]. For each problem specification, we use a neural-network to represent the program itself. The neural-network is used as a *latent* representation of the program. However, this technique is out of the scope of this report, where the objective is to synthesize an interpretable program.

Chapter 4

Program Trees

In this chapter, we explore a real synthesizer, TRINITY [7, 29]. We introduce TRINITY because we will use it as baseline to compare against our synthesizer. TRINITY is an enumerative search based synthesizer that can be instantiated in any domain. To be instantiated in a particular domain it requires two inputs: (1) a DSL, and (2) an interpreter to evaluate syntactically correct programs on that DSL. In the following sections, we explain how TRINITY represents the program space in SMT by encoding a tree-based model, the k-tree model. Subsequently, we present its search algorithm and show how it exploits an off-the-shelf SMT Solver to perform most of the enumerative search. Finally, we empirically evaluate TRINITY by instantiating it in the domain of table transformations. We present and discuss TRINITY's performance on a set of 27 benchmarks comprised of non-noisy input-output examples of tables. The reason we use non-noisy input-output examples is that TRINITY cannot handle noise.

4.1 K-Tree Model

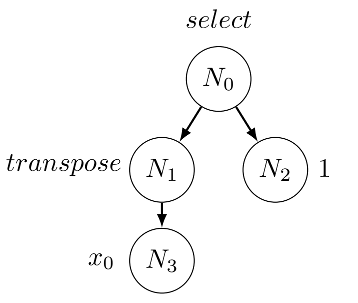
A k-tree of depth d is a tree in which the branching factor is always k, that is, every node but those at maximum depth has k children. We think of k-tree nodes as structures with 3 attributes, (1) a pointer to a DSL element or $null$, (2) a list of pointers to its children, and (3) a unique index. We use s_N to denote the unique index associated with node N. We associate index $s_R = 0$ with the root node R of the k-tree. The index of the j'th child of node N is $s_N \times k + j$.

To show how k-trees can be used to represent programs consider the following scenario. Given an arbitrary DSL, let us fix k as the maximum arity of all DSL components. Any program in a DSL can always be represented by an Abstract Syntax Tree (AST). Moreover, the number of children any AST node of any program is at most k, since k is the maximum arity of all DSL components. Thus, for each AST there is always a corresponding k-tree that represents exactly the same program, except it has some unused nodes assigned to $null$. Therefore, we can always map programs to k-trees.

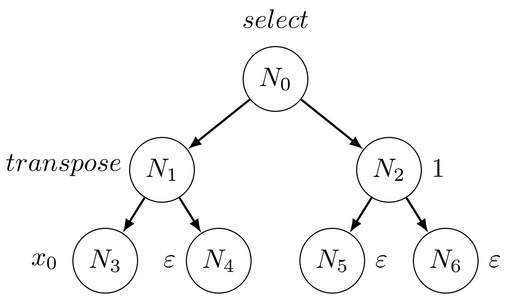
Example 11. Recall the DSL from Example 2, where x_0 is the only program input:

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

The maximum arity of all DSL components is 2. In Figure 4.1(a) we show the AST representation of the program $select(transpose(x_0), 1)$, and in Figure 4.1(b) we show the corresponding 2-tree of depth 2.



(a) AST representation of the program.



(b) 2-tree representation of the program.

Figure 4.1: Representations of the program $select(transpose(x_0), 1)$. We use ε to refer to $null$.

In the k-tree model, the idea is to implicitly represent all the k-trees that correspond to syntactically correct programs using a set of constraints, and then use a constraint solver to enumerate them. One of the motivations to formalize the enumeration process as a constraint solving problem, instead of using a simple graph search formulation, is that constraint solving provides a straightforward way to block programs during the enumeration process. For example, blocking a large set of programs can be as simple as adding an extra constraint. This can be useful if the synthesizer is able to learn generalized patterns of spurious programs (e.g. a particular component does not occur in any solution). In the remainder of this chapter, we shall focus on encoding this model using SMT, resorting only to integer and Boolean variables. The SMT encoding is as follows.

Variables

We want to encode the syntactic structure of programs in SMT using integer variables. Therefore, we must associate each DSL $\mathcal{L}$ element (components, constants, and input variables) with a unique positive integer identifier. For each DSL element e, we use ID(e) to denote the unique positive integer identifier associated with e. Moreover, we say that ID$(null) = 0$. Let us consider an abstract k-tree T of depth d, where k is the maximum arity of all components in $\mathcal{L}$. To encode this tree in SMT, we will have 2 sorts of variables:

- An integer variable v_{s_N} for each node N of the k-tree T, denoting the DSL element assigned to node N;

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

Example 12. The assignment corresponding to the k-tree represented in Figure 4.1(b) is as follows:

$$v_0 = \textsf{ID}(select) \wedge v_1 = \textsf{ID}(transpose) \wedge v_2 = \textsf{ID}(1) \wedge v_3 = \textsf{ID}(x_0) \wedge \bigwedge_{i=4}^{6} v_i = \textsf{ID}(null) \wedge$$
$$b_0 = 1 \wedge b_1 = 1 \wedge \bigwedge_{i=2}^{6} b_i = 0$$

Constraints

Before discussing the constraints, we introduce some notation that will helps us describe them. Given a k-tree T, we use Internals(T), Leaves(T), and Nodes(T) to refer to the internal, leaf, and all nodes of T, respectively[1]. We use Children(N) to refer to children of node N. Given a DSL $\mathcal{L}$, we use Components($\mathcal{L}$), Inputs($\mathcal{L}$), and Constants($\mathcal{L}$) to denote its components, inputs, and constants, respectively. Given a DSL $\mathcal{L}$, We use StartSymbols($\mathcal{L}$) to refer to the components and inputs of $\mathcal{L}$ that can be reached from the starting symbol of $\mathcal{L}$ with just one production rule. Finally, we define the function Domain($C|N = e$), read the domain of node $C \in$ Children(N) given that node N has been assigned to the component e. Domain($C|N = e$) is fully determined by the arguments of component e: (1) if node C is the i'th child of N, and component e has at least i arguments, then Domain($C|N = e$) is the set DSL elements that can be assigned to i'th argument of component e; (2) if node C is the i'th child of N, and component e has less than i arguments, then Domain($C|N = e$) $= \{null\}$. We also say that Domain($C|N = e$) $= \{null\}$ if e is not component. The constraints are as follows.

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

$$\bigvee_{e \,\in\, \textsf{StartSymbols}(\mathcal{L})} v_0 = \textsf{ID}(e) \tag{4.1}$$

Example 13. Consider the DSL from Example 11. Constraint 4.1 on the output node of any 2-tree of this DSL is as follows:

$$v_0 = \textsf{ID}(select) \vee v_0 = \textsf{ID}(transpose) \vee v_0 = \textsf{ID}(x_0)$$

- Constraints asserting that leaf nodes are assigned to either input variables, constants, or $null$:

$$\forall N \in \textsf{Leaves}(T) : \bigvee_{e \,\in\, \textsf{Inputs}(\mathcal{L}) \,\cup\, \textsf{Constants}(\mathcal{L}) \,\cup\, \{null\}} v_{s_N} = \textsf{ID}(e) \tag{4.2}$$

Example 14. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). The constraint

[1] Note that Nodes(T) $=$ Internals(T) $\cup$ Leaf(T). Furthermore, Internals(T) $\cap$ Leaf(T) $= \emptyset$

4.2 on the internal node N_3 is as follows:

$$v_3 = \mathsf{ID}(x_0) \lor \bigvee_{i=1}^{10} v_3 = \mathsf{ID}(i) \lor v_3 = \mathsf{ID}(null)$$

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

$$\begin{gathered}\forall N \in \text{Internals}(T), \forall C \in \text{Children}(N), \forall e_1 \in \text{Components}(\mathcal{L}) : \\ v_{s_N} = \mathsf{ID}(e_1) \implies \bigvee_{e_2 \in \text{Domain}(C|N=e_1)} v_{s_C} = \mathsf{ID}(e_2)\end{gathered} \tag{4.3}$$

Example 15. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). The two instances of constraint 4.3 on the internal node N_1 and component $transpose$ are as follows:

$$v_1 = \mathsf{ID}(transpose) \implies (v_3 = \mathsf{ID}(select) \lor v_3 = \mathsf{ID}(transpose) \lor v_3 = \mathsf{ID}(x_0))$$
$$v_1 = \mathsf{ID}(transpose) \implies v_4 = \mathsf{ID}(null)$$

- Constraints ensuring that if internal nodes are assigned to constants, inputs or $null$, then their children are assigned $null$

$$\begin{gathered}\forall N \in \text{Internals}(T), \forall C \in \text{Children}(N) : \\ \bigvee_{e \in \text{Inputs}(\mathcal{L}) \cup \text{Constants}(\mathcal{L}) \cup \{null\}} v_{s_N} = \mathsf{ID}(e) \implies v_{s_C} = \mathsf{ID}(null)\end{gathered} \tag{4.4}$$

Example 16. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). The two instances of constraint 4.4 on the internal node N_1 are as follows:

$$(v_1 = \mathsf{ID}(x_0) \lor \bigvee_{i=1}^{10} v_1 = \mathsf{ID}(i) \lor v_1 = \mathsf{ID}(null)) \implies v_3 = \mathsf{ID}(null)$$
$$(v_1 = \mathsf{ID}(x_0) \lor \bigvee_{i=1}^{10} v_1 = \mathsf{ID}(i) \lor v_1 = \mathsf{ID}(null)) \implies v_4 = \mathsf{ID}(null)$$

Optional Constraints

The previously defined constraints are crucial to ensure that only syntactically correct programs are encoded into the SMT formula. However, it very useful to encode extra constraints that further reduce the program space. These constraints can be used to eliminate unlikely programs from being explored.

- Constraints stating that each input variable must be used at least once:

$$\forall e \in \text{Inputs}(\mathcal{L}) : \bigvee_{N \in \text{Nodes}(T)} v_{s_N} = \mathsf{ID}(e) \tag{4.5}$$

Example 17. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). Constraint 4.5

for the only input variable x_0 is as follows:

$$v_0 = \textsf{ID}(x_0) \lor v_1 = \textsf{ID}(x_0) \lor v_2 = \textsf{ID}(x_0) \lor v_3 = \textsf{ID}(x_0) \lor v_4 = \textsf{ID}(x_0) \lor v_5 = \textsf{ID}(x_0) \lor v_6 = \textsf{ID}(x_0)$$

- Constraints used to enforce exactly p components to be used:

$$\forall N \in \textsf{Nodes}(T) : \bigvee_{e \,\in\, \textsf{Components}(\mathcal{L})} v_{s_N} = \textsf{ID}(e) \iff b_{s_N} = 1 \tag{4.6}$$

$$\Big(\sum_{N \,\in\, \textsf{Nodes}(T)} b_{s_N} \Big) = p \tag{4.7}$$

Example 18. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). If we want to force the generated program to have 2 components, we add the following constraints:

$$(v_0 = \textsf{ID}(select) \lor v_0 = \textsf{ID}(transpose)) \iff (b_0 = 1)$$
$$(v_1 = \textsf{ID}(select) \lor v_1 = \textsf{ID}(transpose)) \iff (b_1 = 1)$$
$$(...)$$
$$(v_6 = \textsf{ID}(select) \lor v_6 = \textsf{ID}(transpose)) \iff (b_6 = 1)$$
$$(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6) = 2$$

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

$$\bigvee_{N \,\in\, \textsf{Leaves}(T)} \neg(v_{s_N} = \textsf{ID}(null)) \tag{4.8}$$

Example 19. Consider the DSL from Example 11, and the 2-tree of Figure 4.1(b). Constraint 4.8 is as follows:

$$\neg(v_3 = \textsf{ID}(null)) \lor \neg(v_4 = \textsf{ID}(null)) \lor \neg(v_5 = \textsf{ID}(null)) \lor \neg(v_6 = \textsf{ID}(null))$$

4.2 K-Tree Search

4.2.1 Blind Search Algorithm

Now that we have defined and shown how to build a k-tree model for a given DSL, we can use it to construct a simple enumeration based algorithm for a given problem specification. One of the keys to the algorithm is the selection of an adequate depth for the k-tree. Choosing a fixed depth can be a hazard: if the depth is too small we might end up searching in an extremely restricted program space without any solutions. On the other hand, if the depth is too big we might end up compromising optimality (in the sense that we could have found a program of smaller depth) and efficiency (due to the exponential nature of the k-tree model). Alternatively, instead of choosing a fixed depth we can use an iterative method. The idea is that we can start by trying to find a solution of depth 0 and iteratively increase the depth until we can find a solution. The depth is only increased whenever the current program space is exhausted. In Algorithm 4.1 we present a k-tree search-based algorithm that relies on this method.

Algorithm 4.1 works as follows. In each iteration of the **for** loop in lines $1-10$, we explore program spaces of different depths. In line 2, the function call BUILDFORMULA creates the SMT formula of the k-tree using the constraints previously described. It should include the optional constraint 4.8, used to enforce the leaf nodes of the k-tree to be used. Using this optional constraint prevents the same program from being enumerated more than one time at different depths (e.g. enumerating a program of depth 1 when we explore the k-tree of depth 1, and enumerating the same program of depth 1 when we explore the k-tree of depth 2) . In lines $3-9$, we ask the SMT solver to enumerate all programs of the corresponding k-tree. The function CONVERMODEL is used to convert the SMT assignment to the representation of the program in the DSL. The function BLOCKMODEL is used to prevent the same program from being enumerated twice.

```
Algorithm 4.1: SYNTHESIZE($\mathcal{L}, \Lambda, k, n$)
input  : $\mathcal{L}$, a domain specific language
         $\Lambda$, a problem specification
         $k$, the maximum arity of any function in the DSL
         $n$, the maximum depth of the solution
output: $P$, a program in $\mathcal{L}$ of depth at most $n$ that satisfies the problem specification $\Lambda$, or $\emptyset$ if
         there is no such program
1  for $i = 0$ to $n$ do
2      $\mathcal{F} \leftarrow$ BUILDFORMULA($\mathcal{L}, k, i$)         // build the $k$-tree formula
3      $\omega \leftarrow$ SMTSOLVE($\mathcal{F}$)
4      while $\omega \neq \emptyset$ do
5          $P \leftarrow$ CONVERTMODEL($\mathcal{L}, \mathcal{F}, \omega$)   // convert the model to a program
6          if ISCORRECT($P, \Lambda$) then
7              return $P$
8          $\mathcal{F} \leftarrow$ BLOCKMODEL($\mathcal{F}, \omega$)     // prevent enumeration of the same model twice
9          $\omega \leftarrow$ SMTSOLVE($\mathcal{F}$)
10 return $\emptyset$
```

```
Algorithm 4.2: ISCORRECT ($P, \Lambda$)
input  : $P$, a program
         $\Lambda$, a problem specification
output: $1$ if the program $P$ complies with the problem specification $\Lambda$, or $0$ otherwise
1  for $(\vec{x}_i, y_i) \in \Lambda$ do
2      if $P(\vec{x}_i) \neq y_i$ then
3          return $0$
4  return $1$
```

There are three final notes we would like to emphasize regarding Algorithm 4.1. The first is that given a sufficiently large n Algorithm 4.1 always finds a solution if there is one (it enumerates all the k-trees of

```
Algorithm 4.3: BlockModel($\mathcal{F}_{in}, \omega$)
  input : $\mathcal{F}_{in}$, a formula representing a $k$-tree
          $\omega$, the model to be blocked
  output: $\mathcal{F}_{out}$, a formula representing the $k$-tree in which $\omega$ is not a model
1 $\varphi \leftarrow \{\}$
2 for $(x_i = j) \in \omega$ do
3     $\varphi \leftarrow \varphi \cup \{\neg(x_i = j)\}$       // build the constraint $\bigvee_{(x_i=j) \in \omega} \neg(x_i = j)$
4 $\mathcal{F}_{out} \leftarrow \mathcal{F}_{in} \cup \{\varphi\}$         // append the new constraint to the $k$-tree formula
5 return $\mathcal{F}_{out}$
```

depth up to n, hence all programs of depth up to n). The second is that the solutions found by Algorithm 4.1 are always of minimum depth (solutions of depth m are always explored before solutions of depth $m + 1$, for any $0 \leq m \leq n - 1$). The third is that in Algorithm 4.1 iterations of the **for** loop in lines $1 - 10$ are independent of each other. It follows that from an implementation point of view it would be easy to introduce a simple kind of parallelism, as we could always search for solutions of different depth at the same time.

4.2.2 Ordered Search Algorithm

The current SMT encoding of the k-tree model provides us a way to perform a brute force search of the program space. However, as previously discussed, simply performing a blind brute force search usually does not suffice, due to the size of the program space [22]. One way to tackle this problem is to perform a 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 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 [7]. 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 $e_1(e_2(...), ...)$ starts at node N.

Objective Function

We use Occurs($\mathcal{L}$), and IsParent($\mathcal{L}$) 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.

$$\sum_{occurs(e,w) \in \text{Occurs}(\mathcal{L})} w \cdot o_e + \sum_{is\_parent(e_1,e_2,w) \in \text{IsParent}(\mathcal{L})} w \cdot p_{e_1,e_2} \tag{4.9}$$

Example 20. Consider the DSL from Example 11, and the following predicates:

$$occurs(select, 10) \tag{4.10}$$
$$occurs(transpose, 2) \tag{4.11}$$
$$is\_parent(select, transpose, 1) \tag{4.12}$$
$$is\_parent(transpose, transpose, 10) \tag{4.13}$$

The program $select(transpose(x_0), 1)$ satisfies predicates 4.9, 4.10, and 4.11. Therefore, it has a score of $10 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 + 10 \cdot 0 = 13$.

Constraints

We use Internal(T), and Nodes(T) to denote the internal nodes, and all nodes of k-tree T, respectively. We also use Children(N) to denote to children of node N. ID is the function that maps DSL elements 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.

$$\forall occurs(e, w) \in \text{Occurs}(\mathcal{L}) : \bigvee_{N \in \text{Nodes}(T)} v_{s_N} = \text{ID}(e) \iff o_e = 1 \tag{4.14}$$

Example 21. Consider the DSL from Example 11, the 2-tree of depth 2 from Figure 4.1(b), and the predicate $occurs(select, 10)$. Constraint 4.14 for the predicate $occurs(select, 10)$ is as follows:

$$(v_0 = \text{ID}(select) \lor v_1 = \text{ID}(select) \lor v_2 = \text{ID}(select) \lor v_4 = \text{ID}(select) \lor$$
$$v_5 = \text{ID}(select) \lor v_6 = \text{ID}(select)) \iff o_{select} = 1$$

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

$$\forall is\_parent(e_1, e_2, w) \in \text{IsParent}(\mathcal{L}), \forall N \in \text{Internal}(T) :$$
$$v_{s_N} = \text{ID}(e_1) \wedge \Big(\bigvee_{C \in \text{Children}(N)} v_{s_C} = \text{ID}(e_2)\Big) \iff p_{e_1,e_2,N} = 1 \tag{4.15}$$

$$\forall is\_parent(e_1, e_2, w) \in \text{IsParent}(\mathcal{L}) : \bigvee_{N \in \text{Internal}(N)} p_{e_1,e_2,N} = 1 \iff p_{e_1,e_2} = 1 \tag{4.16}$$

Example 22. Consider the DSL from Example 11, the 2-tree of depth 2 from Figure 4.1(b), and the predicate $occurs(select, 10)$. For the predicate $is\_parent(select, transpose, 1)$, constraints 4.15, and 4.16 are as follows:

$$(v_0 = \text{ID}(select) \wedge (v_1 = \text{ID}(transpose) \vee v_2 = \text{ID}(transpose))) \iff p_{select,transpose,N_0} = 1$$
$$(v_1 = \text{ID}(select) \wedge (v_3 = \text{ID}(transpose) \vee v_4 = \text{ID}(transpose))) \iff p_{select,transpose,N_1} = 1$$
$$(v_2 = \text{ID}(select) \wedge (v_5 = \text{ID}(transpose) \vee v_6 = \text{ID}(transpose))) \iff p_{select,transpose,N_2} = 1$$
$$(p_{select,transpose,N_0} = 1 \vee p_{select,transpose,N_1} = 1 \vee p_{select,transpose,N_2} = 1) \iff p_{select,transpose} = 1$$

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 $is\_parent(e_1, e_2, w)$ adds an exponential number of variables and constraints. The more of these predicates we add, the more complex the optimization problem will be.

4.2.3 Experimental Setup

To evaluate the search algorithm's performance, we decided to run a series of benchmarks on a simplified version of our problem. We used an existing *Python 3* implementation of the k-tree model, TRINITY [29]. TRINITY uses the Z3 [5, 9] SMT solver to enumerate programs. TRINITY requires the user to provide both a DSL, and an interpreter capable of performing evaluations on syntactically correct programs. Thus, we designed and implemented a DSL for table manipulation that closely resembles R. The DSL is described in Appendix A. In our DSL every function has an arity of at most 4. Therefore, TRINITY used a 4-tree model to encode the program space.

Subsequently, we built 27 different benchmarks (adapted from MORPHEUS [12] paper), each comprised of (1) a non-noisy input-output example, (2) a hand-made solution (Table 4.1 briefly summarizes the size of these solutions), and (3) a custom DSL. The reason each test case has its own tweaked DSL is that the DSL is fixed throughout the whole synthesis process, meaning that all constant values (strings, column numbers, etc) to be used in the solution must be included in the DSL from the beginning. Therefore, benchmark-specific constants are only included in the necessary benchmarks. Additionally, we used all optional constraints described in section 4.1. We also used the $is\_parent(e_1, e_2, w)$ and $occurs(e, w)$ predicates described in section 4.2.2. The weights of the predicates were obtained by counting the number of times each of them was satisfied in our hand-made solutions.

Finally, there is one key difference between the implementation and the synthesis algorithm. Although the SYNTHESIZE algorithm shown in Algorithm 4.1 is designed to search for a solution by iteratively trying 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 primarily evaluate the performance of the synthesizer on feasible program spaces.

Solution depth	2	3	4
# Tests	6	15	6

Table 4.1: Brief summary of the 27 benchmarks

We ran each benchmark with a time limit of 3600 seconds, and one at time (serially), on a Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz with 64GB of RAM.

4.2.4 Results and Discussion

In Figure 4.2, we show the total execution time of each instance with a timeout of 3600 seconds. The synthesizer was able to solve correctly 14 out of 27 benchmarks (51.85%). We consider a solution correct if there is a one-to-one correspondence between rows of the generated output table and the correct output table. In Figure 4.3, we plot the total number of programs enumerated on each instance, until either a solution is found or a timeout occurs. We can observe there are some instances in which the synthesizer enumerated very few programs, and in some cases (1 to 6) it did not enumerate a single program. The maximum number of programs enumerated within the time limit was 8437. To understand which function call was consuming the most execution time, we used a profiler on one of the benchmarks we could solve. We concluded that approximately 90% of the execution time was taken by a single function call, SMTSOLVE, the function used to enumerate 4-trees.

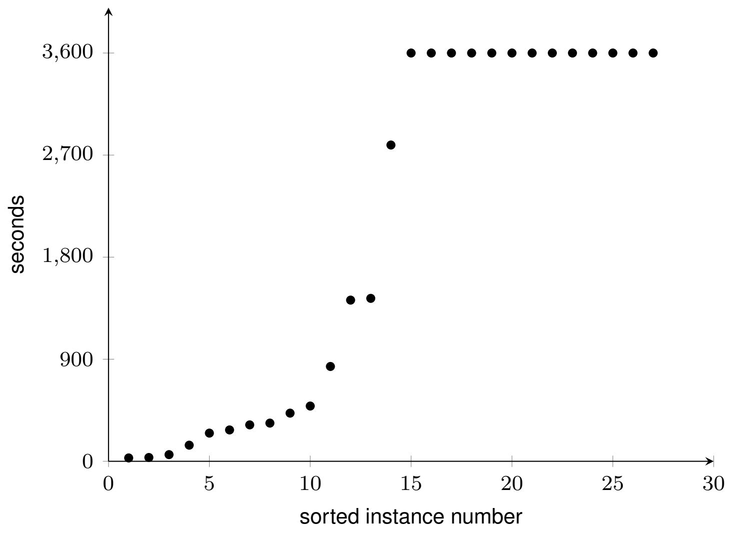


Figure 4.2: Results obtained for tests 1 to 27 using the k-tree model. The total execution time is displayed on the *seconds* axis. The *instance* axis contains the test instances sorted by execution time, meaning $x = 1$ corresponds to the test that took the least amount of time to pass, and $x = 27$ the one that took the most.

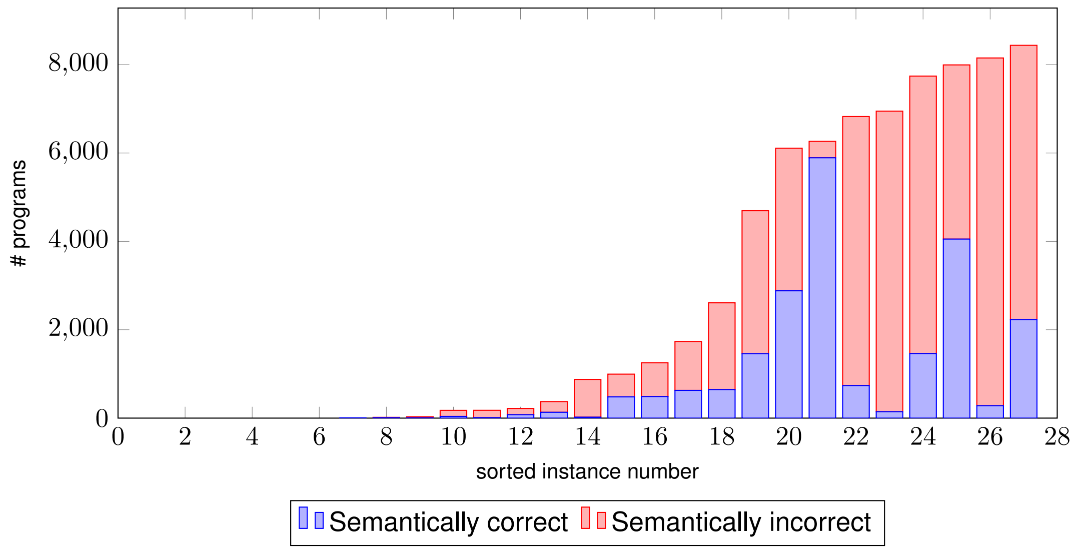


Figure 4.3: Results obtained for tests 1 to 27 using the k-tree model. We show the number of semantically correct and incorrect programs enumerated by the synthesizer with a timeout of 3600 seconds (it stopped either due to reaching a solution or timeout). The instances were sorted according to the total number of programs enumerated.

We conducted a further investigation to try to understand why there were 6 tests in which the synthesizer was not able to enumerate a single program. We found out these tests corresponded to instances in which we enforced the synthesizer to find solutions of depth 4 (the 6 instances of depth 4 pointed out in Table 4.1). The synthesizer was getting stuck on the SMTSOLVE call, without being able to enumerate a single model. The SMT encoding of the 4-tree model is too big for state-of-art SMT solvers. Moreover, we also noticed that although our programs were growing at a constant rate (single line accretions), the model was growing exponentially. Figure 4.4 shows how fast the 4-tree grows, and how many of its nodes are used by programs in which each line only depends on the previous. To represent a program of this kind with 4 lines on a 4-tree of depth 4, we only need to use at most 17 out of its 324 nodes. Data transformation tasks usually resemble this structure, and the unnecessarily large size of the model is compromising our ability to enumerate simple programs.

We have also identified another issue with this model. We can see in Figure 4.3 that the majority of the programs generated by the synthesizer were semantically incorrect. It is often the case that the programs enumerated by the synthesizer, albeit syntactically correct according to DSL, produced interpreter errors when evaluated on the given inputs. These errors arise because the DSL is fixed throughout the synthesis process, and oftentimes the inputs of table manipulation operations cannot be decided until run-time. For example, if we want to select a set of columns from a table T, we first need to know what are the columns of table T. We address these issues in the next chapter.

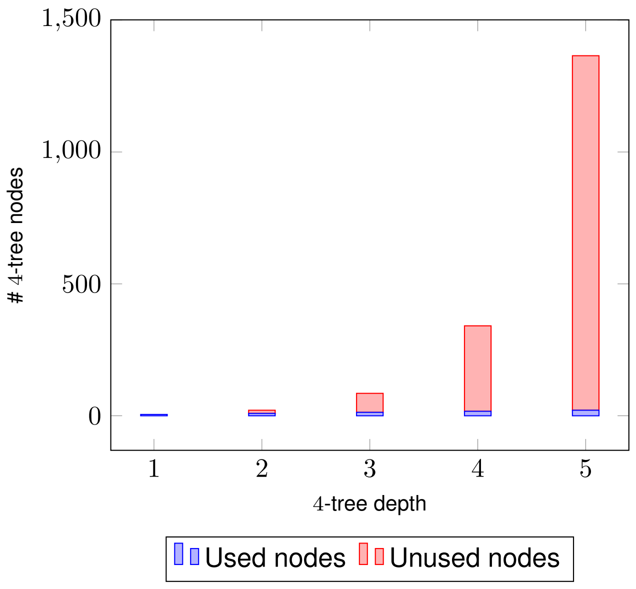


Figure 4.4: Number of 4-tree nodes used to encode a sequential program.

Chapter 5

Sketch Trees

In this chapter, we aim to address the two main problems of the k-tree model we have identified from our empirical evaluation in the previous chapter: (1) the low enumeration ratio (programs per second), and (2) the high percentage of semantically incorrect programs generated throughout the synthesis process.

5.1 Increasing the Enumeration Ratio

In the previous chapter, we have identified and discussed the reason that leads to the low enumeration ratio (programs per second). The k-tree model grows exponentially, and so does the SMT encoding we described. Having such a large model and encoding to represent small programs is hindering the synthesizer's performance, due to the expensive calls to the SMT solver. To overcome this problem, we could use a more compact representation that better suits our problem's structure (i.e. a model to represent programs where each line oftentimes only depends on the previous). For instance, we could achieve this by using a line-based encoding [33] rather than a tree-based encoding. Another alternative is to divide the enumeration of programs into smaller and simpler steps. In this section, we opt to divide the enumeration in a loop consisting of two smaller steps: (1) build a sketch (i.e. an incomplete program comprised of only components), (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 7. (Sketch) Consider a domain specific language $\mathcal{L}$ described by a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where $\Sigma = \mathcal{F} \cup \mathcal{I} \cup \mathcal{C}$[1], 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 23. Recall the DSL from Example 2, where x_0 is the only program input:

$$S \rightarrow select(S, M) \mid transpose(S) \mid x_0$$

$$M \rightarrow 1 \mid 2 \mid 3 \mid ... \mid 10$$

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

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.

The separation of the enumeration of programs in two steps is based on the idea of sketching [46, 47, 48]. Different variants of the two-step enumeration process having been explored by different authors [8, 12, 13, 45, 50, 52, 54]. The difference between all of them is within the two individual steps. For instance, some synthesizers use a type-directed search to perform the sketch completion step [12, 45, 50], whereas others use other synthesizers, such as SKETCH [46], to perform the sketch completion step [8]. Ultimately, the goal is to achieve the reduction in complexity of the enumeration process by dividing it into simpler steps, following the divide and conquer principle. One of our differentiating ideas is within the sketch enumeration step. In particular, we seek to keep the expressiveness the SMT encoding of the k-tree model provides us, while achieving the desired reduction in complexity. The reason we follow this approach is that the SMT representation of the k-tree model allows us to: (1) use predicates (e.g. $occurs(e, w)$, or $is\_parent(e_1, e_2, w)$ provided in section 4.2.2) to seamlessly guide the search, and (2) support learning mechanisms such as NEO [14]'s.

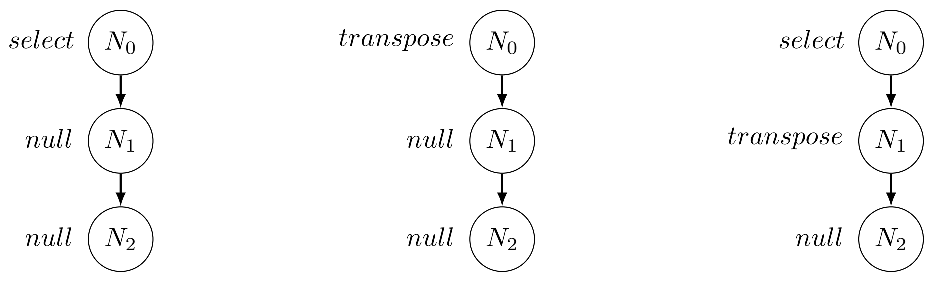
Next, we describe how to perform the sketch enumeration step using a variant of the k-tree model.

5.1.1 K-tree Model for Sketches

The first step in our algorithm requires us to encode the space of syntactically correct sketches. This search space can also be represented using a k-tree, albeit using a much smaller k. Since we only need to represent components in the k-tree, k only has to be the maximum component-arity of all components. The component-arity of a given component e is the number of components e can take as input.

Example 24. Consider the DSL from Example 23. 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 25. Consider the DSL from Example 23. Since the maximum component-arity of all DSL components is 1, a 1-tree suffices to represent all sketches in this DSL. In Figure 5.1, we show some examples of sketches of this DSL represented in a 1-tree of depth 2.



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

Figure 5.1: Various sketches in the DSL of Example 23 represented in a 1-tree of depth 2.

Next, we shall focus on the encoding of the k-tree for sketches in SMT.

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 $null$ ($b_{s_N} = 0$).

Constraints

We will use a notation similar to that of Section 4.1. Given a k-tree T, we use Internals(T), and Nodes(T) to refer to the internal, and all nodes of T, respectively. We use Children(N) to refer to children of node N. Given a DSL $\mathcal{L}$, we use Components$(\mathcal{L})$, and 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 ComponentDomain$(C|N = e)$, read the component domain of node $C \in$ Children(N) given that Node N has been assigned to component e. ComponentDomain$(C|N = e)$ 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 ComponentDomain$(C|N = e)$ 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 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:

$$\bigvee_{c \,\in\, \text{ComponentStartSymbol}(\mathcal{L})} v_0 = \textsc{Id}(c) \tag{5.1}$$

  Example 26. Consider the DSL from Example 23. Constraint 5.1 on the output node of any 1-tree of this DSL is as follows:

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

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

$$\forall N \in \text{Internals}(T), \forall C \in \text{Children}(N), \forall c_1 \in \text{Components}(\mathcal{L}) :$$
$$v_{s_N} = \text{ID}(c_1) \implies \bigvee_{c_2 \in \text{ComponentDomain}(C|N=c_1) \cup \{null\}} v_{s_C} = \text{ID}(c_2) \tag{5.2}$$

Example 27. Consider the DSL from Example 11, and the 1-tree of Figure 5.1(a). Constraint 5.2 on the internal node N_1, for component $transpose$ is as follows:

$$v_1 = \text{ID}(transpose) \implies (v_2 = \text{ID}(select) \lor v_2 = \text{ID}(transpose) \lor v_2 = \text{ID}(null))$$

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

$$\forall N \in \text{Internals}(T), \forall C \in \text{Children}(N) :$$
$$v_{s_N} = \text{ID}(null) \implies v_{s_C} = \text{ID}(null) \tag{5.3}$$

Example 28. Consider the DSL from Example 23, and the 1-tree of Figure 5.1(a). Constraint 5.3 on the internal node N_1 is as follows:

$$v_1 = \text{ID}(null) \implies v_2 = \text{ID}(null)$$

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:

$$\forall N \in \text{Nodes}(T) : v_{s_N} = \text{ID}(null) \iff b_{s_N} = 0 \tag{5.4}$$
$$\Big(\sum_{N \in \text{Nodes}(T)} b_{s_N} \Big) = p \tag{5.5}$$

Example 29. Consider the DSL from Example 23, and the 1-tree of Figure 5.1(a). If we want to enforce the generated program to have 2 components, we add the following constraints:

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

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

$$\bigvee_{N \in \text{Nodes}(T)} \neg(v_{s_N} = \text{ID}(null)) \tag{5.6}$$

Example 30. Consider the DSL from Example 23, and the 1-tree of Figure 5.1(a). If we want to enforce the leaf to be used, we add the following constraint:

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

Besides these constraints, we can also encode into the k-tree formula predicates to guide the search. Since the encoding of these predicates is equal to that of regular k-tree described in section 4.2.2, we will not replicate them here.

5.1.2 Sketch-based Search Algorithm

Now that we have shown how to encode all possible sketches in SMT, we can construct a sketch-based enumeration algorithm. The structure of the synthesis algorithm remains equal to that of SYNTHESIZE shown in Algorithm 4.1, however, enumerating programs now involves more than just a simple call to the SMT solver. Recall that our objective is to divide the enumeration of programs in two steps: (1) sketch enumeration, and (2) sketch completion. The sketch enumeration can still be achieved by using an off-the-shelf SMT solver and the encoding described in the previous subsection. Although the sketch completion could also be encoded in SMT, in this section, we explore a graph-based method for this step. 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, but have their domain limited according to the component assigned to their parent. Example 31 illustrates this data-structure for a particular sketch.

Example 31. Consider the DSL of Example 23 and the sketch $select(transpose(S), M)$. Figure 5.2 illustrates the incomplete AST of this sketch. Note that the domain of leaf nodes is comprised of either constant values or input variables. If a leaf child of a node cannot be assigned to neither a constant nor an input variable, we consider its domain empty and represent it by $\{\}$.

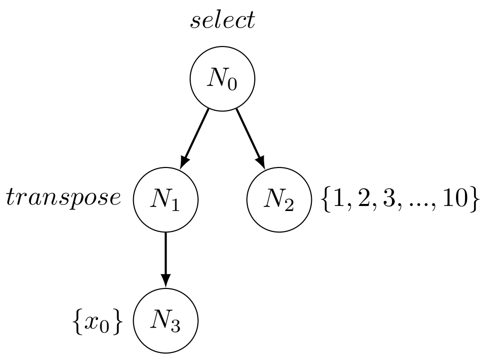


Figure 5.2: Incomplete AST for the sketch $select(transpose(S), M)$.

Algorithm 5.1 shows the new synthesis algorithm SKETCHSYNTHESIZE that leverages both the k-tree formula for sketches, and the data structure previously described. Similarly to the previous synthesis algorithm, we explore program spaces of different depths incrementally. In line 2, we start by building the k-tree formula for sketches. The formula should include the optional constraint 5.6, 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 3, we construct the sketch's incomplete AST by calling the function BUILDTREE in line 5. Subsequently, in the **for** loop of lines $6-9$, we sweep through all possible completions of the incomplete AST, and test them against the problem specification. In line 6, we start by calling the function FILLTREE(T), which returns a list with all possible completions of T. In lines $7-9$, we convert each tree into a program in the DSL, and test it against the problem specification.

Algorithm 5.2 shows the FILLTREE(T) function that returns a list with all completions of the incomplete AST with root T. FILLTREE resembles a depth-first search. If T is a leaf node, we return a list with all possible assignments to T, each one representing a different tree. If T is not a leaf node, we recursively call FILLTREE on children of T. The result of calling FILLTREE on each child of T is a list of all possible complete AST starting at the corresponding child node. After we go through the **for** loop of lines $8-9$, each element of $child\_trees$ is a list with all possible complete AST starting at one of the child nodes. The function call COMBINATIONS$(child_trees)$ returns a list where each element is a combination of sub-trees. For example, suppose that $child\_trees$ is the list $[[t_1, t_2], [t_3, t_4], [[t_5]]$. The possible combinations are COMBINATIONS$(child_trees) = [\{t_1, t_3, t_5\}, \{t_1, t_4, t_5\}, \{t_2, t_3, t_5\}, \{t_2, t_4, t_5\}]$. In the **for** loops of lines $11-12$, we sweep through every combination of sub-trees of different children. Essentially, what we are doing is picking a sub-tree of each child, in order to build the base of a bigger tree with root node T.

Algorithm 5.1: SKETCHSYNTHESIZE$(\mathcal{L}, \Lambda, k, n)$

```
input : $\mathcal{L}$, a domain specific language
        $\Lambda$, a problem specification
        $k$, the maximum component-arity of any component in the DSL
        $n$, the maximum depth of the solution
output: $P$, a program in $\mathcal{L}$ that satisfies the specification $\Lambda$ of depth at most $n$, or $\emptyset$, if there is no
        such program)
1  for $i = 0$ to $n$ do
2      $\mathcal{F} \leftarrow$ BUILDSKETCHFORMULA$(\mathcal{L}, k, i)$      // build the $k$-tree formula for sketches
3      $\omega \leftarrow$ SMTSOLVE$(\mathcal{F})$                         // $\omega$ is a sketch
4      while $\omega \neq \emptyset$ do
5          $T \leftarrow$ BUILDTREE$(\mathcal{L}, \mathcal{F}, \omega)$           // convert the sketch to an incomplete AST
6          for $T_c \in$ FILLTREE(ROOT$(T)$) do       // for every completion of $T$
7              $P \leftarrow$ TREETOPROGRAM$(T_c)$
8              if ISCORRECT$(P, \Lambda)$ then
9                  return $P$
10         $\mathcal{F} \leftarrow$ BLOCKMODEL$(\mathcal{F}, \omega)$             // block the current sketch
11         $\omega \leftarrow$ SMTSOLVE$(\mathcal{F})$
12 return $\emptyset$
```

Algorithm 5.2: FILLTREE(T)

```
input  : $T$, an AST node
output: $trees$, a list with all possible completions with root $T$
if IsLeaf($T$) then
    $trees \leftarrow []$
    for $v_i \in$ Values($T$) do
        $trees \leftarrow trees + [\{T = v_i\}]$  // appending the possible assignments to a list
    return $trees$
else
    $child\_trees \leftarrow []$                    // a list of lists of complete trees
    for $c \in$ Children($T$) do
        $child\_trees \leftarrow child\_trees +$ [FillTree($c$)] // all completions starting at $c$
    $trees \leftarrow []$
    for $combination \in$ Combinations($child\_trees$) do // all possible combinations of
        $trees \leftarrow trees + [combination \cup \{T = T.v\}]$  // sub-trees of different children
    return $trees$
```

5.1.3 Results and Discussion

To test the new SKETCHSYNTHESIZE algorithm we decided to run a set of benchmarks. The experimental setup used was identical to that of section 4.2.3. We used the same set of benchmarks, all optional constraints, and the predicates $occurs(e, w)$ and $is\_parent(e_1, e_2, w)$. We also forced the synthesizer to search for a solution of depth equal to our hand-made solution. Since maximum component-arity in our DSL is 2, we used 2-trees to represent the sketch space.

In Figure 5.3, we compare the run-times obtained using this approach with the run-times using the k-tree model. We are now able to solve four more instances, a total of 18 out of 27 test cases (66.66%). In Figure 5.4, we plot the total number of programs enumerated for each instance, until either a solution is found or timeout occurs. Using this approach, we are able to enumerate programs for every instance, even for those at depth 4. Comparing the number of programs enumerated using the sketches method (Figure 5.4) with the number of programs enumerated using the k-tree model (Figure 4.3), we can see that we are now able to enumerate up to $10\times$ more programs in the same amount of time, which is compliant with the results we discussed earlier: if we were to consider the execution time of each test excluding the SMTSOLVE calls, we would have enumerated $10\times$ more programs[2].

Although we achieved great progress using this approach in terms of execution time, the number of semantically incorrect programs is still too high. We can see in Figure 5.4 that the number of semantically incorrect programs vastly outnumbers the semantically correct programs. We will address this issue in the following section.

[2]Using the previous approach, we were able to enumerate up to 8437 programs in 3600 seconds. Moreover, 90% of the execution time was taken by SMTSOLVE calls. Therefore, in theory, if we were to continue enumerating programs for 3600 seconds without considering the time taken by SMTSOLVE calls, we would have enumerated 84370 programs.

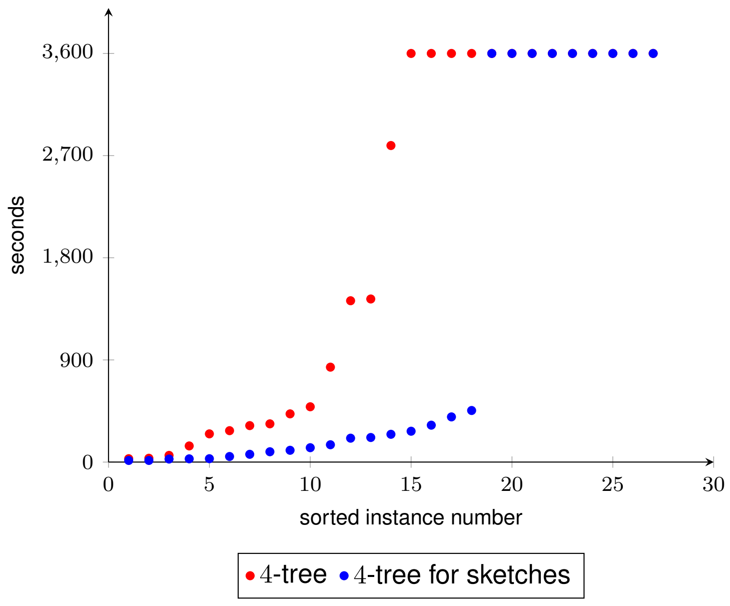


Figure 5.3: Results obtained for tests 1 to 27 using the k-tree for sketches and static constants. The total execution time is displayed on the *seconds* axis. The *instance* axis contains the test instances sorted by execution time, meaning $x = 1$ corresponds to the test that took the least amount of time to pass, and $x = 27$ the one that took the most.

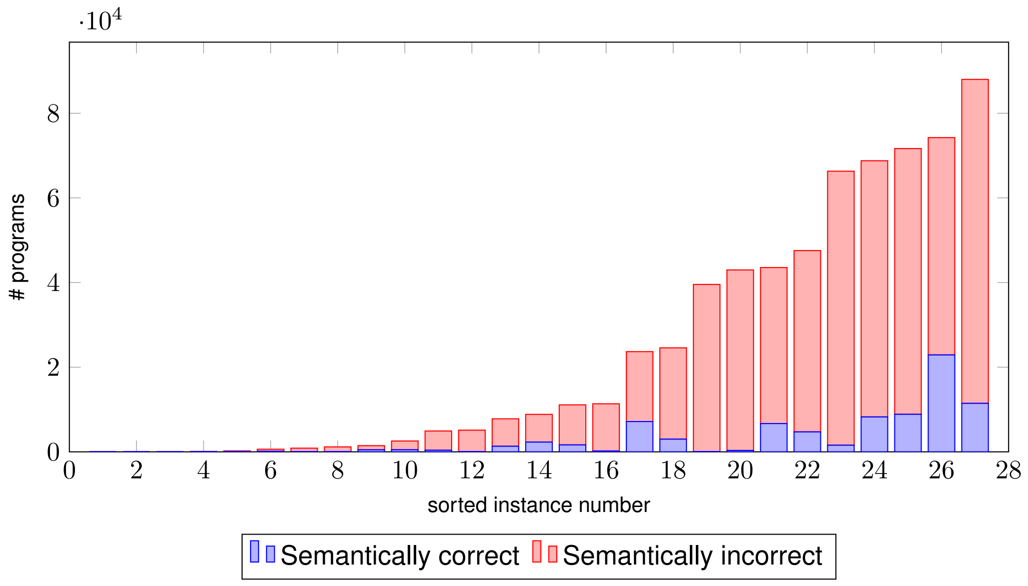


Figure 5.4: Results obtained for tests 1 to 27 using the k-tree for sketches and static constants. We show the number of semantically correct and incorrect programs enumerated by the synthesizer with a timeout of 3600 seconds (it stopped either due to reaching a solution or timeout). The instances were sorted according to the total number of programs enumerated.

5.2 Enumerating Semantically Correct Programs

Enumerating programs from a static DSL that does not take context into consideration has driven us to the problem of generating semantically incorrect programs. To illustrate this statement let us consider our target domain: table manipulation operations. These operations are often dependent on the structure of the table (i.e. the number of rows and columns), its column types (i.e. columns of integers, strings), or even the content of its rows. For example, consider that we want to select columns of a given table. It makes sense to restrict the synthesizer to only consider column numbers lower than the total number of columns of the table, since we cannot select a column that does not exist. However, our current approach always uses the same set of constants, engraved on the DSL the user has provided. Thus, always using the same set of constants for select on different tables might result in semantic errors. Example 32 illustrates some syntactically correct programs in a DSL that generate semantic errors in the given problem specification.

Example 32. Consider the DSL from Example 23, and the following program specification comprised of one input table and one output table:

Table 5.1: Input table of Example 31.

1	2	3
4	5	6
7	8	9

Table 5.2: Output table of Example 31.

1	4	7
2	5	8
3	6	9

Although the programs $select(x_0, 4)$, $select(x_0, 4)$, ..., and $select(x_0, 10)$ are syntactically correct according to the DSL, they will generate semantic errors if evaluated on the provided input table (because these programs are selecting columns that do not exist).

5.2.1 Dynamic Constants

In order to tackle this issue, we propose a way to dynamically adapt the DSL to the program being synthesised. We will still use the k-tree model for sketches, but the way we complete the sketches will be different. In essence, we will maintain the structure of SKETCHSYNTHESIZE shown in Algorithm 5.1, but we will be changing the FILLTREE function in line 6. Our idea is the following. In order to prevent the generation of semantically incorrect programs, we need to more carefully select the DSL elements we assign to leaf nodes during the sketch completion step. In particular, the set of DSL elements available for each leaf node should depend not only on the component assigned to its parent node, but also on the assignments made to the leaf node's siblings. We can achieve this if we assign DSL elements to leaf nodes of a component wholly at once. Moreover, the sibling nodes corresponding to components should be evaluated to actual values, since these values can be crucial in deciding the set of available DSL elements. Example 33 illustrates our idea.

Example 33. Consider the DSL from Example 2, where x_0 is the only input of the program:

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

Consider that the input-output example is comprised of an input table with 3 rows and 6 columns. Suppose the synthesizer enumerated the sketch $select(transpose(S, M)$. In Figure 5.5(a), we show the incomplete AST for this sketch. Node N_3 is a leaf node but has no siblings, thus its domain is still computed using the static DSL. Since N_2 is a leaf node with one sibling N_1, its domain should be computed dynamically. To compute N_2's domain, we need to evaluate its sibling node corresponding to a component. In Figure 5.5(b), we show the new AST after we assign N_3 to x_0, and after we evaluate the sub-tree N_1 (t_0 is the table generated after performing the evaluation). Using t_0 and knowing that N_0 has been assigned to $select$, we can compute the domain of N_2. Since t_0 has 3 columns, the domain of N_2 is $\{1, 2, 3\}$.

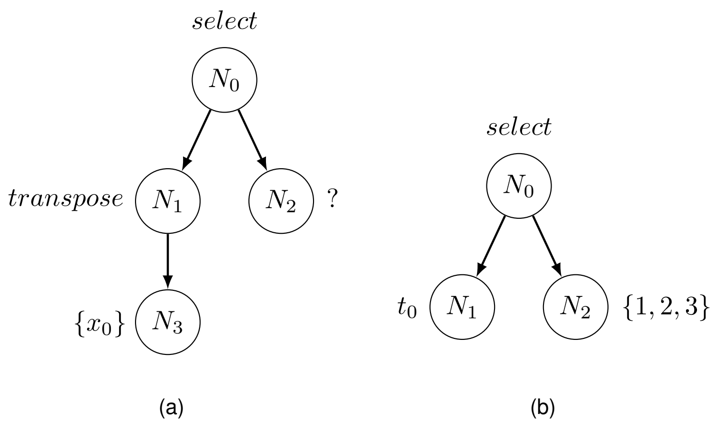


Figure 5.5: Incomplete AST of the sketch $select(transpose(M), S)$, and the incomplete AST of the same sketch after performing an evaluation.

The DYNAMICFILLTREE function in Algorithm 5.3 illustrates a high-level implementation of this idea. In lines $1 - 3$, we recursively call DYNAMICFILLTREE on the non-leaf children of the input node. In lines $7 - 10$, we go through every combination of sub-trees of the non-leaf nodes. In line 8, we call the function FILLLEAVES, which returns a list with all possible assignments to the leaf children of an internal node given the assignments to its non-leaf siblings and its parent. For instance, consider Example 33 and the AST from Figure 5.5(a). Calling FILLLEAVES on the leaf children of node N_0 yields the following list: FILLLEAVES($[N_2], [N_0 = select, N_1 = transpose, N_3 = x_0\}) = [\{N_2 = 1\}, \{N_2 = 2\}, \{N_3 = 3\}]$.

5.2.2 Results and Discussion

In order to test this idea, we modified the SKETCHSYNTHESIZE algorithm to use the new DYNAMICFILL-TREE algorithm. The experimental setup was identical to that of section 5.1.3. The only difference is that we are now using a generic DSL for all examples, since constant values are computed dynamically.

Algorithm 5.3: DYNAMICFILLTREE(T)

```
input  : $T$, an AST node
output: $trees$, a list with all possible completions with root $T$
1  $child\_trees \leftarrow []$          // a list of lists of complete trees
2  for $c \in$ INTERNALCHILDREN($T$) do          // the non-leaf children of $T$
3      $child\_trees \leftarrow child\_trees +$ [DYNAMICFILLTREE($c$)]   // all completions starting at $c$
4  if $child\_trees = []$ then            // if we reach a node whose children are only leaves
5      $child\_trees \leftarrow [[null]]$       // we add a dummy value to get a single combination
6  $trees \leftarrow []$
7  for $combination \in$ COMBINATIONS($child\_trees$) do
8      $constants\_list \leftarrow$ FILLLEAVES(LEAFCHILDREN($T$), $combination \cup \{T = T.v\}$)
9      for $constants \in constants\_list$ do
10         $trees \leftarrow trees + [combination \cup constants \cup \{T = T.v\}]$
11 return $trees$
```

Figure 5.6 illustrates how this approach compares to the previous approach. We are now able to solve an additional instance, but there are four instances in which the performance has degraded. The reason is that we are now performing a lot more evaluations using the interpreter, due to the constants being computed dynamically, which can slow down the enumeration process. The speedup on the remaining instances is due to the fact that we can now discard sketches faster, because we do not have to exhaustively search on sketches that will necessarily produce semantic errors.

In Figure 5.7 we plot the total number of programs enumerated for each instance until either a solution is found or a timeout occurs. Although using this approach we enumerate fewer programs than with the previous approach (Figure 5.4), all of the enumerated programs are semantically correct. Since we no longer generate semantically incorrect programs, we use the stacked bars of Figure 5.7 to illustrate the reasons the generated programs were deemed incorrect. We do three checks (in order) before considering a program correct: (1) we compare the structure of the output table produced by the program (number of rows and columns) with the structure of the correct output table; (2) we check if the column types of the produced output table and the column types of the correct output table match; (3) we check if there is a one to one correspondence between the rows of the produced table and the correct output table. Figure 5.7 shows the stage at which we dismissed the programs as incorrect. We can see that most of the programs we enumerate produce output tables with the incorrect structure. Previous work [12] has shown that we could greatly benefit from using pruning techniques to discard spurious incomplete programs, since sometimes it is possible to identify structural errors on incomplete programs.

In Figure 5.8 we show the total number of sketches enumerated until either a solution is found or a timeout occurs. The synthesizer was only able to explore up to 132 different sketches in 3600 seconds. This reinforces our reason to guide the search through the use of predicates, as we cannot waste time exploring bad sketches.

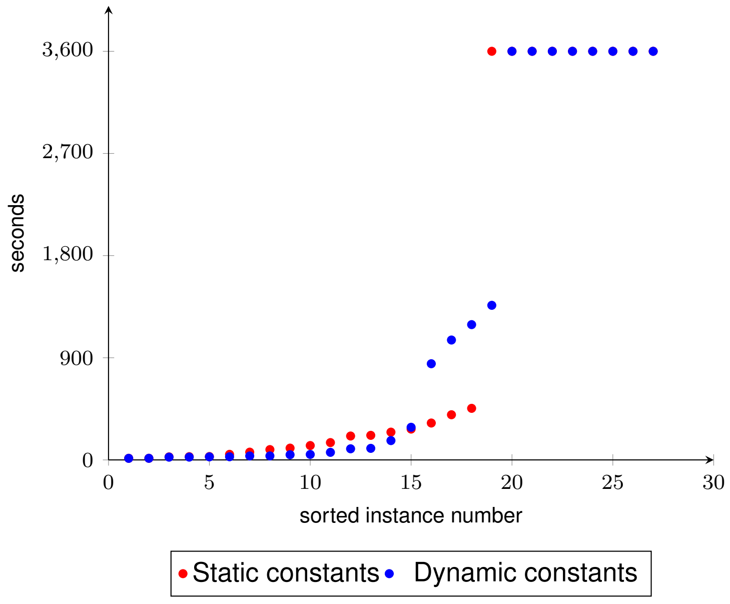


Figure 5.6: Results obtained for tests 1 to 27 using the k-tree for sketches and dynamic constants. The total execution time is displayed on the *seconds* axis. The *instance* axis contains the test instances sorted by execution time, meaning $x = 1$ corresponds to the test that took the least amount of time to pass, and $x = 27$ the one that took the most.

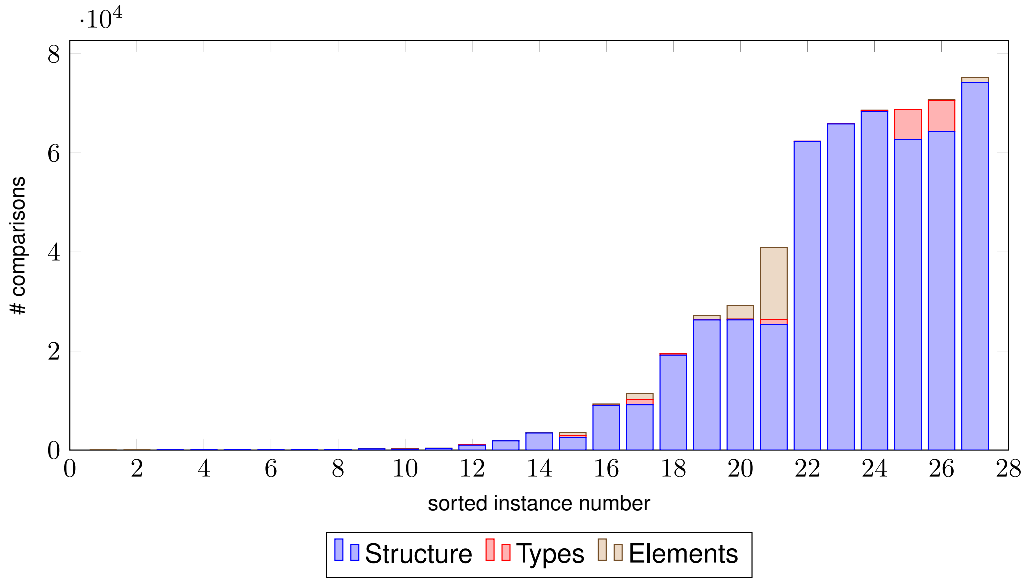


Figure 5.7: Results obtained for tests 1 to 27 using the k-tree for sketches and dynamic constants. We show the total number of programs the synthesizer enumerated with a timeout of 3600 seconds (it stopped either due to reaching a solution or timeout). The instances were sorted according to the total number of programs enumerated. The stacked bars represent the number of programs discarded at each stage.

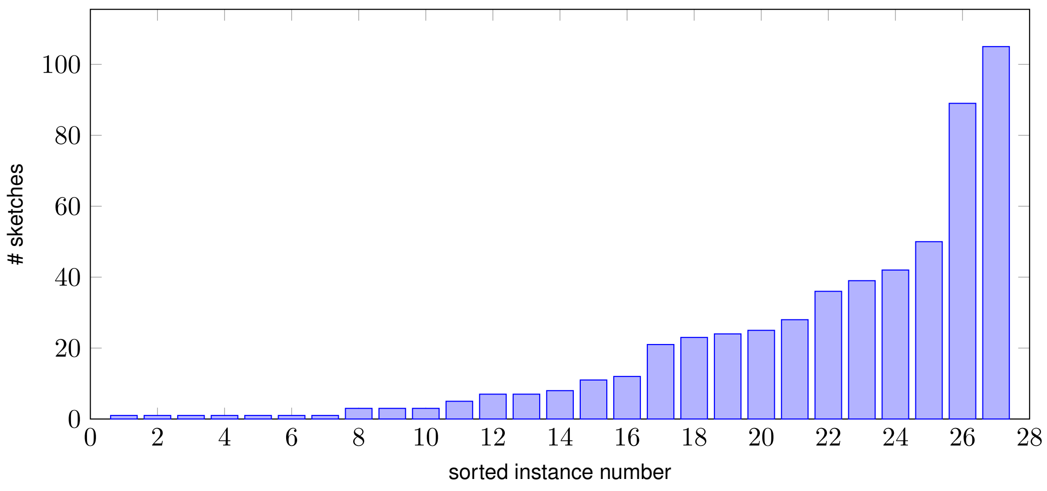


Figure 5.8: The number of sketches explored in each instance within the 3600 seconds time limit. The test instances were sorted according to the total number of sketches enumerated.

Chapter 6

Synthesizing with Noise

In chapter 4, we discussed the workings of a state-of-art program synthesizer. Subsequently, we modified it in order to better fit our needs. Through the usage of a dynamic DSL, and the smaller k-tree model for sketches, we achieved a significant performance increase over the original synthesizer in the domain of table transformations. We sought these performance improvements because synthesizing programs from noisy tabular data is a generalization of the problem of synthesizing from regular tabular data. Therefore, if we want to leverage existing work and ideas to solve our problem, we had to first adapt them to the problem of synthesizing programs from regular input-output examples of tables.

In this chapter, we focus on synthesizing programs from noisy examples. The classical formulation of program synthesis is to find a program that satisfies exactly every input-output example. However, in this case, we do not want to find a program that satisfies the specification exactly, but a program providing a good generalization of the input-output behavior. Problem statement 2 shows a possible formulation of this problem.

Problem statement 2. Given a domain specific language $\mathcal{L}$, a problem specification Λ with 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 $P = \operatorname{argmin}_{P' \in \mathcal{L}} \; c(P', \Lambda)$.

6.1 Generalized Program 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 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, similarly to what is done in SYNTHESIZE shown in Algorithm 4.1. 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 6.1, we show a generalized synthesis algorithm COSTSYNTHESIZE that uses this method.The structure of the algorithm is similar to that of SKETCHSYNTHESIZE shown in Algorithm 5.1. The only difference is the stopping criterion, whereas SYNTHESIZE returns a program whenever it finds one that satisfies the problem specification, COSTSYNTHESIZE goes through the entirety of the program space.

```
Algorithm 6.1: COSTSYNTHESIZE($\mathcal{L}, c, \Lambda, k, n$)
  input  : $\mathcal{L}$, a domain specific language
           $\Lambda$, 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 $P = \mathrm{argmin}_{P' \in \mathcal{L}}\ c_\Lambda(P')$
1  $P_{best} \leftarrow \emptyset$
2  for $i = 0$ to $n$ do
3      $\mathcal{F} \leftarrow$ BUILDSKETCHFORMULA($\mathcal{L}, k, i$)    // build the $k$-tree formula for sketches
4      $\omega \leftarrow$ SMTSOLVE($\mathcal{F}$)                      // $\omega$ is a sketch
5      while $\omega \neq \emptyset$ do
6          $T \leftarrow$ BUILDTREE($\mathcal{L}, \mathcal{F}, \omega$)          // convert the sketch to an incomplete AST
7          for $T_c \in$ DYNAMICFILLTREE(ROOT($T$)) do   // for every completion of $T$
8              $P \leftarrow$ TREETOPROGRAM($T_c$)
9              if $c(P, \Lambda) \leq c(P_{best}, \Lambda)$ then
10                 $P_{best} \leftarrow P$
11         $\mathcal{F} \leftarrow$ BLOCKMODEL($\mathcal{F}, \omega$)          // block the current sketch
12         $\omega \leftarrow$ SMTSOLVE($\mathcal{F}$)
13 return $P_{best}$
```

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[1]. 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, as discussed in chapter 4. To implement this non-optimal version of the synthesis algorithm, the only modification we have to make to COSTSYNTHESIZE is to add a stopping criterion (e.g. 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.

[1]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.

(a)

$col1$	$col2$
“3.5”	3
“4”	3
“10”	1

(b)

$col1$	$col2$
Smith	0.2
Jones	0.6
Williams	0.1
Taylor	0.1

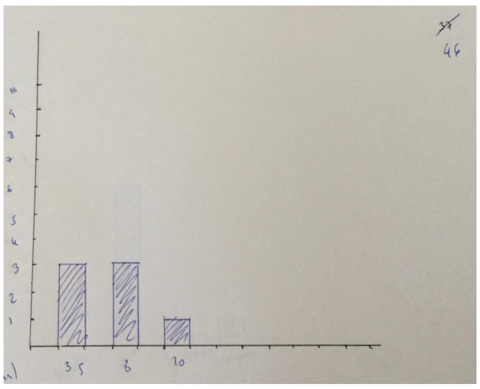
(c)

$col1$	$col2$
Rusty	10
Horatio	7
Bob	3

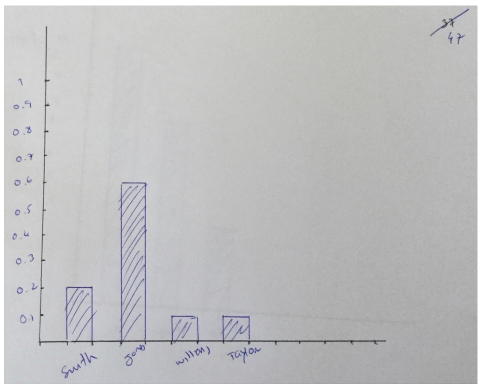
(d)

$col1$	$col2$
“1”	33
“3”	25.5
“7”	35
“8”	25.5
“9”	40
“10”	16

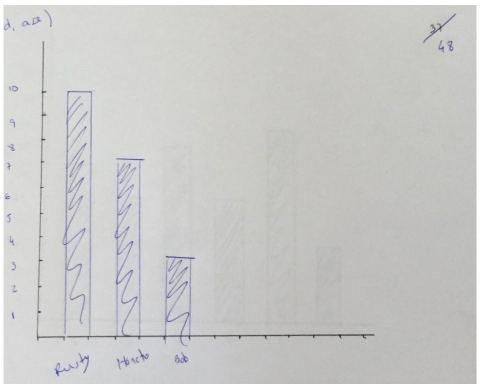
Figure 6.1: Real output tables corresponding to four examples.



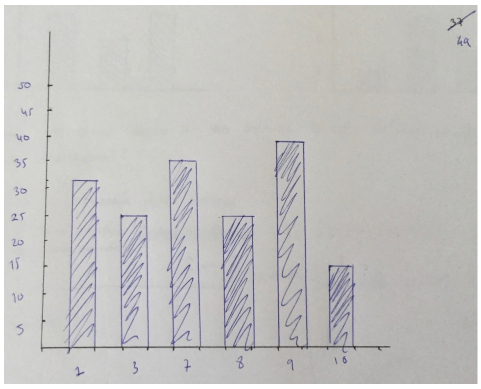
(a)



(b)



(c)



(d)

Figure 6.2: Hand-drawn bar plots corresponding to the four examples of Figure 6.1.

(a)

$col1$	$col2$
“3.5”	3.13687
“4”	3.28771
“10”	1.17598

(b)

$col1$	$col2$
Smith	0.21114
Jones	0.62300
Williams	0.10872
Taylor	0.10436

(c)

$col1$	$col2$
Rusty	10.02932
Horatio	7.36156
Bob	3.40391

(d)

$col1$	$col2$
“1”	32.02550
“3”	25.14164
“7”	36.10481
“8”	25.65155
“9”	39.67422
“10”	16.34560

Figure 6.3: Obtained output tables corresponding the four hand-drawn plots of Figure 6.2.

6.1.1 Cost Function

The cost function we use has to reflect the underlying noise model of the problem we want to solve. In our case, we want to synthesize programs from input-output examples of tables, in which the output tables can have numerical errors. Therefore, the question we must answer is which kind of numerical error do we want to support in our tables, and how can we reflect it onto the cost function. To get insights into a real noisy environment (and thus create a realistic noise model), we decided to hand-draw plots from 25 real input-output examples adapted from a well-known databases' book [39]. For each input-output example, we drew on a sheet of paper the corresponding output table in a bar plot using a linear scale. Each output table had 2 columns, (1) a column of strings (the x-axis), and (2) a column of positive numerical values (the y-axis). In Figure 6.1 we show four of these output tables, and in Figure 6.2 the corresponding bar plots. Subsequently, we used WebPlotDigitizer[2], a web-based tool to extract data from images, to recover the numerical data from the hand-drawn plots. Finally, we replaced the numerical data of the original output tables with the numerical data obtained with the tool. In Figure 6.3 we show the new output tables corresponding to the plots of Figure 6.2. In this way, we generated 25 input-output examples with the characteristics we sought.

Going through the process of extracting the numerical data from each hand-drawn plot we noticed that the tool's accuracy was not dependent on the scale being used. For example, in Figure 6.2(b) we can see that the scale being used is from 0 to 1 (y's axis minimum and maximum values). If we look at the corresponding correct output table in Figure 6.1(b), and the obtained output table of the same example in Figure 6.3(b), we can see that the maximum row-wise distance between corresponding values is $|0.6 - 0.623| = 0.023$ (second row). On the other hand, in Figure 6.2(d) we can see that the scale being used is from 0 to 50 (y's axis minimum and maximum values). If we look at the corresponding correct output table in Figure 6.1(d), and the obtained output table of the same example in Figure 6.3(d), we can see that the maximum row-wise distance between corresponding values is $|35 - 36.10481| = 1.10481$ (third row). However, if normalize this value to a value between 0 and 1 using the chart's maximum and minimum values we get $\frac{|35-36.10481|}{50-0} \times 100 = 0.022096$. In fact, in our examples, the normalized row-wise distance for all tables was always a value in the interval $[0, 0.1]$. Considering these observations, in Algorithm 6.2, we propose a cost function to compare tables with two columns, a column of categorical values, and a column of numerical values.

Algorithm 6.2 works as follows. In lines $3 - 7$, we verify if the output table from the specification y_i, and the output table generated $(\hat{y_i})$ have a similar structure. If the tables do not have the same number of rows or columns, or if they do not have columns with matching types (for example if y_i has 2 columns of integers, and $\hat{y_i}$ has 2 columns of strings), we consider the program incorrect. In lines $8-11$, we verify if the categorical columns of both tables are equal (we sort the column values because the row order does not matter). In lines $12 - 15$, we sort the rows of both tables, firstly according to their categorical value, and then according to their numerical value (to break ties). In lines $16 - 19$, we verify if there is any pair of rows whose normalized distance deviates more than ϵ. Lastly, at line 21 we return the sum of the deviation from input-output examples of the problem specification.

[2]https://www.automeris.io/WebPlotDigitizer/

Algorithm 6.2: COSTFUNCTION(P, Λ)

```
input  : $P$, a program
         $\Lambda$, a program specification
output: $c$, the cost of program $P$ on specification $\Lambda$
$c \leftarrow 0$
for $(\vec{x_i}, y_i) \in \Lambda$ do
    $\hat{y_i} \leftarrow P(\vec{x_i})$
    if $\text{NROWS}(\hat{y_i}) \neq \text{NROWS}(y_i)$ or $\text{NCOLS}(\hat{y_i}) \neq \text{NCOLS}(y_i)$ then
        return $+\infty$  // if tables have a different number of rows or columns
    if $\text{TYPES}(\hat{y_i}) \neq \text{TYPES}(y_i)$ then
        return $+\infty$  // if tables have different column types (ex int,int vs str,int)
    $categorical_1 \leftarrow \text{SORT}(\text{CATEGORICALCOL}(\hat{y_i}))$        // sorted alphabetically
    $categorical_2 \leftarrow \text{SORT}(\text{CATEGORICALCOL}(y_i))$        // sorted alphabetically
    if $categorical_1 \neq categorical_2$ then
        return $+\infty$  // if categorical columns are different, then program is wrong
    $y_i \leftarrow \text{SORT}(y_i, key = lambda\ x : \text{CATEGORICAL}(x), \text{NUMERICAL}(x))$
    $\hat{y_i} \leftarrow \text{SORT}(\hat{y_i}, key = lambda\ x : \text{CATEGORICAL}(x), \text{NUMERICAL}(x))$
    $numerical_1 \leftarrow \text{NUMERICALCOL}(\hat{y_i})$
    $numerical_2 \leftarrow \text{NUMERICALCOL}(y_i)$
    for $i = 0$ to $i = \text{NROWS}(y_i)$ do
        $d \leftarrow \left| \frac{numerical_1[i] - numerical_2[i]}{\text{MAXVALUE}(y_i) - \text{MINVALUE}(y_i)} \right|$    // normalizing distance to the interval of $[0, 1]$
        if $d > \epsilon$ then
            return $+\infty$ // if there is a deviation of more than $\epsilon$, discard the program
        $c \leftarrow c + \frac{d}{\text{NROWS}(y_i)}$
return $c$
```

6.1.2 Experimental Setup

In order to evaluate the new COSTSYNTHESIZE algorithm shown in Algorithm 6.1, along with the proposed COSTFUNCTION shown in Algorihtm 6.2, we decided to run a series of benchmarks. The benchmarks were automatically generated, because there was no data available for our problem (since the problem itself is a novelty). Furthermore, generating a large set of benchmarks using the process described in section 6.1.1 is a very manual process, since it involves drawing charts with some precision. Therefore, to generate the benchmarks we used the following process:

1. We start by generating an input table using the following process:

   (a) Uniformly sample the number of columns in the interval $[2, 4]$, and assign them a type (either a column of strings, or a column of integers);

   (b) Uniformly sample the number of rows in the interval $[1, 10]$;

   (c) Generate rows by sampling from a pool of numbers and letters, from 0 to 255, and a to m, respectively (we enforce letters to repeat with a probability in order to generate groups);

Example 34. In Figure 6.4, we show four input tables generated using the process previously described.

(a)

$col1$	$col2$
d	115
d	86
d	197

(b)

$col1$	$col2$	$col3$
l	186	127
k	19	35
j	247	85
d	194	169

(c)

$col1$	$col2$
a	224
h	143
d	8
e	174
a	58
h	139
d	159
e	209

(d)

$col1$	$col2$
d	66
e	135
c	174
g	234
m	49
j	122

Figure 6.4: Examples of randomly generated input tables.

2. Next, we want to guarantee that there is a program that solves the instance to be generated. Hence, we choose a random program depth d from the interval $[3, 6]$. Afterwards, we randomly sample from the DSL programs of depth d with exactly d different components using the synthesizer, until either we find one program that yields an output table with one column of strings and one column of numerical data, or timeout occurs (600 seconds). If we cannot find a program with the desired properties within 600 seconds, we repeat the process from step 1.

   Example 35. In Figure 6.5, we show four output tables generated using the process previously described using the input tables of Figure 6.4.

(a)

$col1$	$col2$
d	115.0
d	86.0
d	197.0

(b)

$col1$	$col2$
d	2.0
j	2.0
k	2.0
l	2.0

(c)

$col1$	$col2$
a	1.0
a	1.0

(d)

$col1$	$col2$
d	66.0
e	135.0
c	174.0
g	234.0
m	49.0
j	122.0

Figure 6.5: Output tables generated using the input tables of Figure 6.4.

3. We add noise to the numerical data of the generated output table using the follow process:

   (a) For each column c of numerical data, find the maximum value $M(c)$;

   (b) For each numerical value v in the column c, change v to a random value in the interval $[max(0, v - 0.10 \cdot M(c)),\ v + 0.10 \cdot M(c)]$

   Example 36. In Figure 6.6, we show four noisy output tables generated with the process previously described using the output tables of Figure 6.6.

Repeating this process we generated 487 benchmarks, each one comprised of: (1) an input table, (2) a noisy output table, (3) and a correct output table. In Table 6.1, we present a summary of the

(a)

$col1$	$col2$
d	97.27
d	78.12
d	195.52

(b)

$col1$	$col2$
d	2.18
j	1.94
k	2.12
l	2.04

(c)

$col1$	$col2$
a	0.91
a	1.08

(d)

$col1$	$col2$
d	54.03
e	142.02
c	157.62
g	226.98
m	46.66
j	120.17

Figure 6.6: Noisy output tables generated using the output tables of Figure 6.5.

benchmarks. The DSL we used is described in Appendix A. We forced the synthesizer to search for a solution of depth equal to that of the generated solution, instead of going through all depths. In order to guide the synthesizer towards likely solutions, we also encoded $occurs(e, w)$ and $is\_parent(e_1, e_2, w)$ predicates described in section 4.2.2. These predicates were computed using an approach similar to the approach of DEEPCODER [2]. The idea is that we can use our benchmark generation procedure to "train" the synthesizer. Since our benchmark generation procedure also provides solutions, we can count the number of solutions in which each possible predicate is satisfied. To apply this procedure, we generated 200 benchmarks specifically for this purpose (a different set from the 487 used for testing). Using these benchmarks, we generated $occurs(e, w)$ and $is\_parent(e_1, e_2, w)$ predicates with $0 \leq w \leq 200$. Since $is\_parent(e_1, e_2, w)$ predicates are very expensive to encode, we discarded those with w less than 20. Finally, we used the threshold $\epsilon = 0.10$ for the cost function (line 18 of Algorithm 6.2).

Solution Depth	2	3	4	5
# Tests	164	147	101	75

Table 6.1: Brief summary of the 487 generated benchmarks.

We ran each benchmark run on a Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz with 64GB of RAM for 3600 seconds. After the timeout of 3600 seconds occurred, the synthesizer returned the program with minimum cost.

6.1.3 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 benchmarks, 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). We decided to look further into the cases wherein the synthesizer enumerated a correct solution but failed to select it. We observed that the difference between the cost of the selected solutions and correct solutions was rather small. Manually looking into the input-output examples and the solutions selected by the synthesizer, we noticed that the synthesizer was performing slight changes to the input table before doing the correct transformation, thereby overfitting the solution to the noisy output table. In Example 37, we illustrate a specific scenario.

Example 37. Consider the following benchmark with solution $select(x_0, \{col1, col2\})$, that selects both columns $col1, col2$:

$col1$	$col2$	$col3$
l	186	127
k	19	35
j	247	185
d	194	169

Table 6.2: Input table

$col1$	$col2$
l	186
k	19
j	247
d	194

Table 6.3: Correct output table

$col1$	$col2$
l	189.69574
k	40.63835
j	245.05057
d	178.95056

Table 6.4: Noisy output table

Consider the synthesizer generates the program $select(unite(col4, col2, col3, \text{“.”}), \{col1, col4\})$, that creates a new column, $col4$, by uniting the values of $col2$ with $col3$, separating them with a dot, and then selects columns $\{col1, col4\}$. Running this program on the input table yields the following output table:

$col1$	$col4$
l	186.127
k	19.35
j	247.185
d	194.169

Table 6.5: Obtained output table

Calculating the distance of the correct solution by measuring the distance between the correct output table shown in Table 6.3 and the noisy output table shown in Table 6.4, we get $|186 - 189.69574| + |19 - 40.63835| + |247 - 245.05057| + |194 - 178.95056| = 42.33296$. Similarly, if we calculate the distance of the overfitted output table shown in Table 6.5 to the noisy output table shown in Table 6.3, we get $|186.127 - 189.69574| + |19.35 - 40.63835| + |247.185 - 245.05057| + |194.169 - 178.95056| = 42.20996$. Since the distance of the noisy output table to the overfitted output table is smaller, the synthesizer will choose the overfitted solution.

In Figure 6.7, 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.

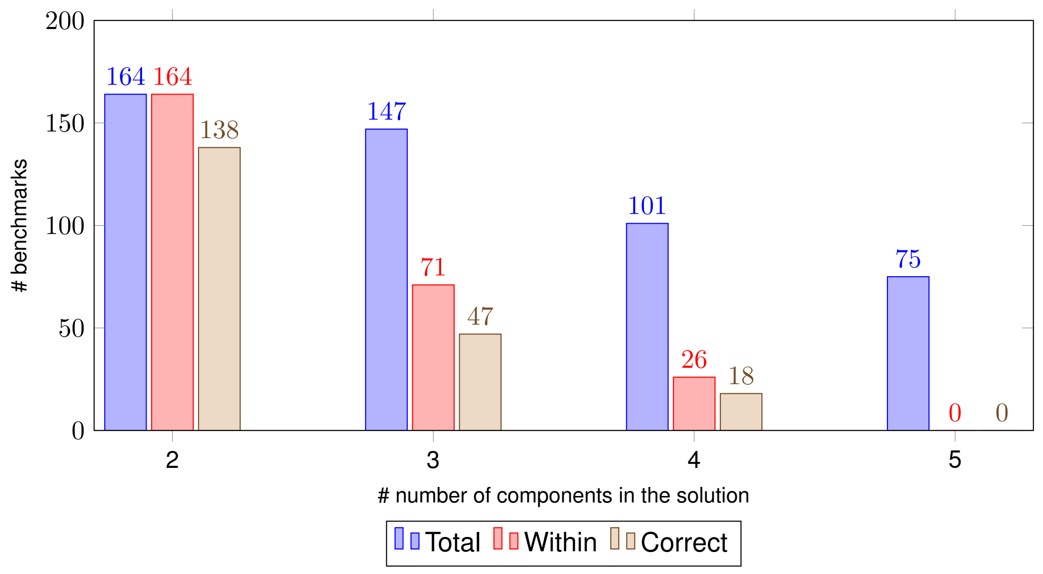


Figure 6.7: Number of solved benchmarks by depth within the time limit of 3600 seconds. For each depth, the blue bar represents the total number of benchmarks with the given depth. The red, and yellow bars represents the number of benchmarks in which a correct program was enumerated, and in which a correct program selected by the synthesizer, respectively.

6.2 Pruning Tecnhiques

In chapter 2, we showed how we could use the DSL Semantics to prune incomplete programs by building a first-order formula encoding properties of that incomplete program (the program specification). We achieved this through the use of the DSL Semantics, that is, a relation that maps each component to a first-order formula describing its high-level properties. For example, using the function $y = filter(x, predicate)$ on a table never increases its number of rows, thus $rows(y) \leq rows(y)$ describes one property of filter that always holds, independently of what is $predicate$. The program specification of a given program P is constructed by composing the abstract properties of the individual components of P. Subsequently, for each example, we would test the consistency between the proprieties of the input-output example, and the relation input-output relation given by the program specification.

Through our empirical evaluation of the synthesis algorithm in the previous chapter, we noticed that the synthesizer was generating many programs that produced tables with the wrong structure. For instance, generating programs that reduce the number of columns of the input table, when we seek to increase them. Previous work [12, 14, 15] suggests that we could see huge performance improvements from the use of pruning techniques, since they help us detect these kinds of spurious programs early. Therefore, to study the impact of pruning, we decided to implement the deductive method for pruning described in chapter 2. We changed the DYNAMICFILLTREE function described in Algorithm 5.3 to test for inconsistencies. The idea is to evaluate the completed sub-trees into actual values, thereby constructing a smaller program in which completed parts are replaced by “new inputs”. Subsequently, we build the program specification of the new smaller program, and test it against the output abstract properties. As a result, we can prune incomplete programs during the program's construction. In Example 38, we illustrate this process.

Example 38. Consider the DSL from Example 2, where x_0 is the only program input:

$$\begin{aligned} S &\to \mathit{select}(S, M) \mid \mathit{transpose}(S) \mid x_0 \\ M &\to 1 \mid 2 \mid 3 \mid ... \mid 10 \end{aligned}$$

A DSL Semantics Ψ of $\mathcal{L}$ is a mapping each of its components to a first-order formula over its input and output variables:

$$\begin{aligned} \Psi(\mathit{select}) &:= \mathit{rows}(y) = \mathit{rows}(x_0) \wedge \mathit{cols}(y) < \mathit{cols}(x_0) \\ \Psi(\mathit{transpose}) &:= \mathit{rows}(y) = \mathit{cols}(x_0) \wedge \mathit{cols}(y) = \mathit{rows}(x_0) \end{aligned}$$

Suppose the problem specification has one I/O Example (x_0, y) with the following properties:

$$\mathit{cols}(x_0) = 10 \wedge \mathit{rows}(x_0) = 3 \wedge \mathit{rows}(y) = 3 \wedge \mathit{cols}(y) = 3$$

Consider the synthesizer is exploring the incomplete program $\mathit{select}(\mathit{select}(x_0, M), M)$. In order to search for inconsistencies it builds the program specification of this program, and tests it against the I/O abstract properties:

$$\begin{aligned} &\mathit{rows}(v_0) = \mathit{rows}(v_1) \;\wedge\; \mathit{cols}(v_0) < \mathit{cols}(v_1) \;\wedge \\ &\mathit{rows}(v_1) = \mathit{rows}(v_3) \;\wedge\; \mathit{cols}(v_1) < \mathit{cols}(v_3) \;\wedge \\ &v_3 = x_0 \;\wedge\; v_0 = y \;\wedge\; \mathit{rows}(x_0) = 3 \;\wedge \\ &\mathit{cols}(x_0) = 10 \;\wedge\; \mathit{rows}(y) = 3 \;\wedge\; \mathit{cols}(y) = 3 \end{aligned}$$

Since this formula is satisfiable, the synthesizer continues to build the program. Suppose the synthesizer decided to assign the inner most M to the constant value 1, thereby generating $\mathit{select}(\mathit{select}(x_0, 1), M)$. Since the inner most $\mathit{select}(x_0, 1)$ has been completed, it can be evaluated into an actual table t_0. The new intermediate table t_0 has the same number of rows as x_0, but only one column:

$$\mathit{rows}(t_0) = 3 \wedge \mathit{cols}(t_0) = 1$$

After evaluating part of the program to t_0, the synthesizer's new task is to complete the program $\mathit{select}(t_0, M)$. Repeating the same process, before assigning a cost value to M, the synthesizer builds the program specification for the new incomplete program, and tests it against the I/O abstract properties:

$$\begin{aligned} &\mathit{rows}(v_0) = \mathit{rows}(v_1) \;\wedge\; \underline{\mathit{cols}(v_0) < \mathit{cols}(v_1)} \;\wedge \\ &\underline{v_1 = t_0} \;\wedge\; \underline{v_0 = y} \;\wedge\; \mathit{rows}(t_0) = 3 \;\wedge \\ &\underline{\mathit{cols}(t_0) = 1} \;\wedge\; \mathit{rows}(y) = 3 \;\wedge\; \underline{\mathit{cols}(y) = 3} \end{aligned}$$

Since this formula is unsatisfiable, the synthesizer can discard the program without completing it.

6.2.1 Results and Discussion

To study the impact of pruning on our problem, we decided to run a set of benchmarks. The experimental setup was identical to that of section 6.1.2. We used the same 487 benchmarks, and the same DSL including the optional constraints and the predicates.

In Figure 6.8, we show the number of benchmarks in which a correct solution was enumerated at some stage, with and without pruning. We can see that the pruning has a significant impact on the synthesizer, mainly on solutions of bigger depth. Using pruning techniques we are able to enumerate correct solutions in 300 out of 487 benchmarks. Discarding incomplete spurious incomplete programs is leaving the synthesizer with more time to explore potentially correct solutions.

In Figure 6.9(a), we compare the times at which the first correct program was enumerated, with and without pruning. In general, we can see that the first correct solution is being enumerated earlier when pruning is used. Moreover, 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 pruning. The reason is that the DSL Semantics we use is not 100% correct. For instance, the abstract specification we use for the $select$ component says that it always reduces the number of columns of tables. However, it is possible (but undesirable) to select all columns of a table using our DSL. We use the DSL Semantics to prevent such cases from happening. It is important to note that this mechanism does not discard all correct solutions, only those with identity operations (e.g. $f(x) = x$). In fact, had we searched for a solution on a program space of smaller depth for each of the 4 cases, we would have found correct solutions.

In Figure 6.9(b), we plot the total number of programs enumerated within the time limit of 3600 seconds, with and without pruning. There are 38 benchmarks in which we are only able to enumerate programs when using pruning. On the other hand, there are also 5 benchmarks in which we are only able to enumerate programs when not using pruning. Moreover, in general, we can see that we are enumerating fewer programs when we use pruning. The reason is that we are not allowing programs with identity operations to be enumerated through the pruning mechanism (i.e. we discard them as spurious programs).

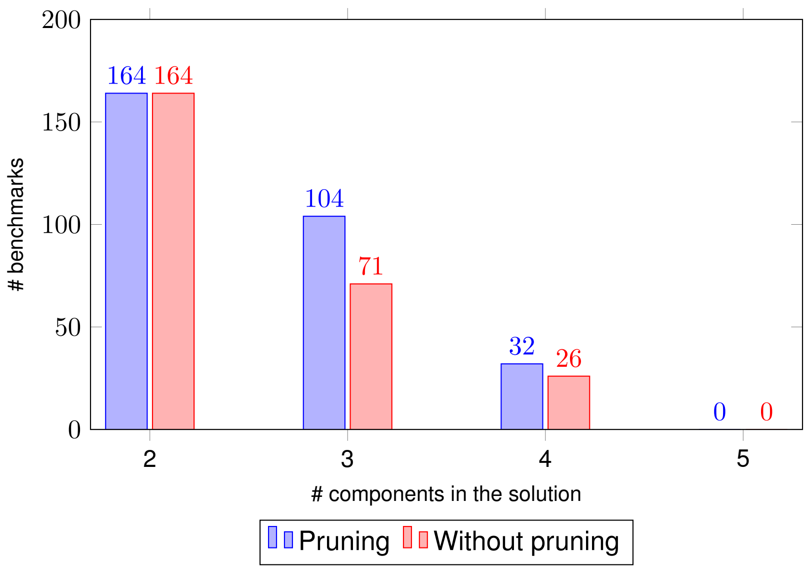
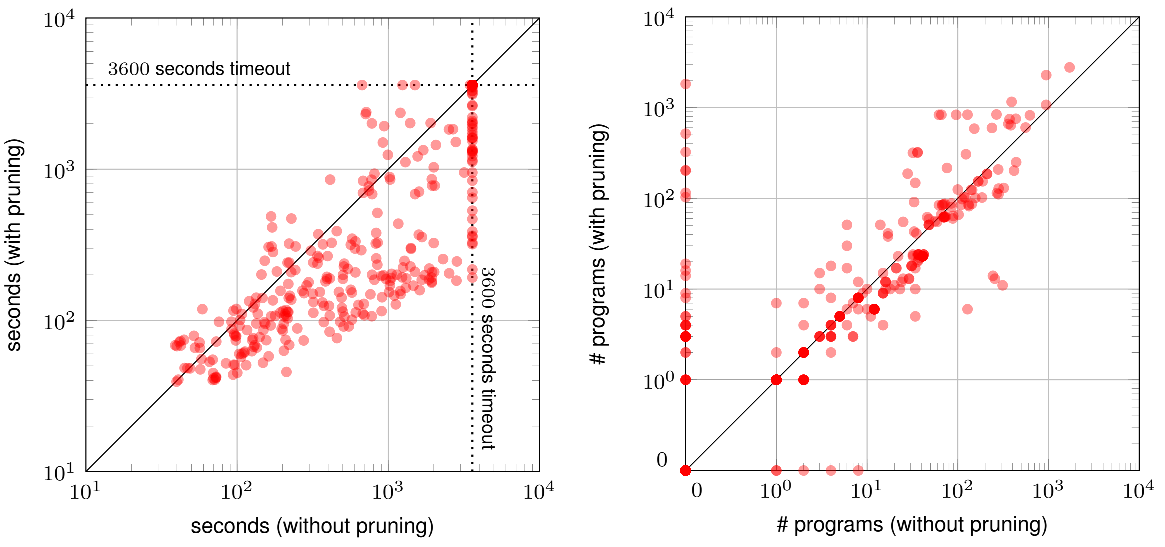


Figure 6.8: Number of benchmarks in which a correct solution was enumerated within the time limit of 3600 seconds. The colored bars represents the number of benchmarks in which a correct solution was enumerated at some point within the time limit, when using pruning (blue bars), and when not using pruning (red bars).



(a) Time at which the first correct solution was enumerated for all benchmarks within the time limit of 3600 seconds.

(b) Number of programs with finite cost enumerated within the time limit of 3600 seconds.

Figure 6.9: Results comparing the synthesizer's performance when using pruning and when not using pruning.

Chapter 7

Conclusions

In this thesis, we addressed a variety of topics in program synthesis. We studied TRINITY [29], a state-of-art synthesizer that uses SMT to enumerate programs. TRINITY implicitly represents the program space by encoding a k-tree in SMT. We formalized this encoding, and showed how we could seamlessly guide the enumeration process through the usage of predicates.

Subsequently, we proposed an enumeration algorithm based on the idea of sketches that divides the enumeration of programs in two steps: (1) sketch enumeration, and (2) sketch completion. We showed how we could use the k-tree model to do the sketch generation step in SMT, thereby preserving the expressiveness that the SMT language provides us. We also proposed two different ways to do the sketch completion step: (1) using static constants that are embedded in the DSL, or (2) using dynamic constants that are computed by performing partial evaluations of programs. The benefits of using sketch enumeration with dynamic constants are two-fold: (1) it does not require the user to tweak the DSL every time they need to synthesize a program, and (2) it only enumerates semantically correct programs.

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 using an heuristic within a fixed time 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 a pruning technique has in our problem. Finally, we built a real program synthesizer and integrated it into the TRINITY [29] tool, which can be used by anyone who wishes to do it[1].

7.1 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, Shi et al. [43] proposed a synthesis algorithm that can achieve a throughput of 31,400 programs per second. We intend

[1]The code is available at https://github.com/danieltrt/Tyrell

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 during the synthesis process, 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, thereby discarding the incorrect program.

There is also some recent work that makes use of other sources of information to reduce the program space. For example, DRACO [32] is a recent program synthesizer for the *Vega-lite* [42] 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.

Bibliography

[1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[2] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

[3] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 218–228, 2015.

[4] P. Bielik, V. Raychev, and M. T. Vechev. PHOG: probabilistic model for code. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2933–2942. JMLR.org, 2016.

[5] 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.

[6] R. Bloem and N. Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, 2010. IEEE. ISBN 978-1-4577-0734-6. URL `https://ieeexplore.ieee.org/xpl/conhome/5766311/proceeding`.

[7] 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.

[8] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013.

[9] 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.

[10] 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.

[11] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 990–998, 2017.

[12] 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.

[13] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

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

[23] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[24] 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.

[25] D. Jurafsky and J. H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. ISBN 9780135041963.

[26] T. A. Lau, S. A. Wolfman, P. M. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[27] 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.

[28] W. Lee, K. Heo, R. Alur, and M. Naik. Accelerating search-based program synthesis using learned probabilistic models. In J. S. Foster and D. Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 436–449. ACM, 2018.

[29] 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. URL `http://www.vldb.org/pvldb/vol12/p1914-martins.pdf`.

[30] S. A. McIlraith and K. Q. Weinberger, editors. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-*

18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Oxleans, Louisiana, USA, February 2-7, 2018, 2018. AAAI Press.

[31] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 187–195. JMLR.org, 2013.

[32] 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.

[33] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, pages 583–599, 2019.

[34] 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.

[35] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

[36] J. Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984. ISBN 978-0-201-05594-8.

[37] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. 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 522–538, 2016.

[38] 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.

[39] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[40] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 419–428, 2014.

[41] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016.

[42] 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.

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

[44] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26. ACM, 2013.

[45] 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.

[46] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008.

[47] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294, 2005.

[48] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415, 2006.

[49] 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.

[50] 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.

[51] X. Wang, I. Dillig, and R. Singh. Synthesis of data completion scripts using finite tree automata. *PACMPL*, 1(OOPSLA):62:1–62:26, 2017.

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

[53] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450, 2017.

[54] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. R. Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 588–594, 2018.

[55] 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.

Appendix A

Domain Specific Language

The DSL we used for the benchmarks is subset of the R programming language[1]. Despite being a subset of R, the DSL is extremely expressive, containing components from the $dplyr$[2], $tidyr$[3], $compare$[4], and $tibble$[5] libraries. It also contains custom components. For readability purposes, we omit both the inputs and the constants from the DSL. Equations A.1, A.2, A.3 show the components we used from $tidyr$ and $dplyr$. The custom components are displayed on Equation A.4.

$$S \rightarrow unite(S) \mid separate(S) \mid spread(S) \mid gather(S) \tag{A.1}$$

$$S \rightarrow filter(S) \mid summarize(S) \mid join\_by(S, S) \mid group\_by(S) \tag{A.2}$$

$$S \rightarrow mutate(S) \mid count(S) \mid select \mid rowwise(S) \tag{A.3}$$

$$S \rightarrow normalize(S) \mid add(S) \mid subtract(S) \mid divide(S) \mid multiply(S) \tag{A.4}$$

We provide a brief description of each component in Tables A.1, A.2, and A.3. Further descriptions of the operations and their concrete implementation can be found on our tool[6].

Function:	Description
$unite(S)$	Combines two columns of table S into one column. The united values can be separated by a special character (e.g. “.”, “_”)
$separate(S)$	The reverse of unite. It separates a column of S into several columns. The separation criterion is a special character (e.g. “.”, “_”).
$gather(S)$	Create two new columns by transforming rows of S into key-value pairs
$spread(S)$	The reverse of gather. It takes two columns of S, one of keys, one of values, and creates a set of new columns, each one corresponding to a different key.

Table A.1: $dplyr$’s components

[1] https://www.cran.r-project.org/
[2] https://www.dplyr.tidyverse.org/
[3] https://www.tidyr.tidyverse.org/
[4] https://www.cran.r-project.org/web/packages/compare/index.html
[5] https://www.tibble.tidyverse.org/
[6] https://www.github.com/danieltrt/Tyrell

Function:	Description
$filter(S)$	Filters the rows of table S according to a predicate.
$summarize(S)$	Operation used to summarize data. It allows us to calculate the mean, max, min, sum of groups of table S.
$join\_by(S_1, S_2)$	Joins two tables by the specified columns of tables S_1, and S_2.
$group\_by(S)$	Creates groups of elements according to the specified columns of table S.
$mutate(S)$	Creates a new column obtained by mutating an existing column of table S.
$count(S)$	Creates a new column with the number of elements of each group of table S.
$select(S)$	Selects a set of columns of table S.
$rowwise(S)$	Creates a group for each element of table S. The number of rows is the number of groups.

Table A.2: $tidyr$'s components

Function:	Description
$normalize(S)$	Rescales a numerical column of table S to the range of 0 to 1.
$add(S)$	Creates a new column by row-wise adding two numerical columns of S.
$subtract(S)$	Creates a new column by row-wise subtracting two numerical columns of S.
$divide(S)$	Creates a new column by row-wise diving two numerical columns of S.
$multiply(S)$	Creates a new column by row-wise multiplying two numerical columns of S.

Table A.3: Custom components