Pure Function Synthesis in the OutSystems Platform

Catarina Coelho
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
Portugal
catarina.a.coelho@tecnico.ulisboa.pt

ABSTRACT

Program synthesis consists in automatically generating a program from a specification used to define user intent. The OutSystems platform is a low-code development platform which allows the development of applications through a graphical user interface. The OutSystems platform allows business logic to be implemented through action flows, which can be used to perform several complex and recurrent operations, such as data wrangling operations. In order to do this, pure functions can be used within OutSystems language expressions to perform these operations. Pure functions are a type of functions that have no side-effects and their returned value is determined by its inputs. However, writing this type of functions might become a tedious and repetitive task due to its recurrence, and might even be a difficult task for less experienced users. In this work we present PUFS, a pure function synthesizer that given a set of input-output examples, as a specification of the function's desired behavior, synthesizes a pure function. Our solution consists of a combination between program sketches as a representation of a partial function and enumeration-based search alongside Satisfiability Modulo Theories (SMT) to fill the sketches in order to obtain the complete function. The proposed solution was evaluated on a set of real-world examples, showing promising results for recurrent and common pure functions.

Author Keywords
Program Synthesis, Satisfiability Modulo Theories, Programming-by-Example

INTRODUCTION

OutSystems platform is a low-code development platform which allows the development of applications through a graphical user interface. Its main goal is to provide an easier and faster experience in development and integration of web and mobile applications. The OutSystems platform allows the implementation of business logic using actions which can be used later in other action flows. An action flow is a set of operations represented by nodes, such as access to a database, assignment of variables, among others, that implements the logic of the application. Unlike regular action flows, action flows can be used within OutSystems language expressions, making them a special case of these type of flows, thus very useful to perform complex data transformations that are recurrent throughout the application. Pure functions are a type of functions that have no side-effects, where the return value is only determined by its input values, as in functions in traditional programming languages. Although the platform provides an easier experience that abstracts the user from the code writing task, it also relies on the use of action flows to prevent the user from having to repeat the same operations. Since the implementation of pure functions in these flows is a frequent element in every application, it makes sense to develop an automation of this process. Code generation has been one of the main recurrent research fields throughout the years, its relevance has become higher and led to the appearance of new research fields and techniques, one of them being program synthesis. Program synthesis consists in automatically generating a program that satisfies a specification provided by the user to express its intent, i.e., the desired behavior of the program. It becomes clear that this technique can be quite useful in the context of our problem, since we want to be able to facilitate the generation of these functions in the platform by some sort of automation, given that these are recurrent tasks throughout the platform. One of the main properties of pure functions is that their output is conditioned by the input. Hence, we can see that the inputs have a great influence in the behavior of these functions. As such, we have chosen an approach based on input-output examples as specification. However, pure functions in the action flows are not code fragments, and so they do not have the structure of typical functions as in other programming languages. These functions are represented as a flow of several nodes in which each node has an operation to be executed within the function. Hence, another reason that implies that synthesis might turn this process easier, since its representation is not as intuitive to automate as other code representations. Due to the regularity of performing these tasks, as well as the characteristics that this type of functions has, it makes sense to automatize this process using synthesis techniques.

2. FUNDAMENTAL CONCEPTS

This chapter provides a brief description of the fundamental concepts required in order to fully understand the rest of this document. Some concepts about Program Synthesis in section 2.1 are presented,
such as its definition, dimensions and main challenges. Then, this chapter introduces and provides some insights on Satisfiability Modulo Theories in section 2.2.

2.1 Program Synthesis

Given a specification used to express the user intent, program synthesis is the task of automatically generating a program that satisfies that specification. Different types of specification include input-output examples [1, 2, 6, 9, 20, 24, 31], logical formulas [14, 16] and natural language [2, 5, 31].
In program synthesis there are three main dimensions, as illustrated in Figure 2.1: expressing user intent, program space and search techniques, which are described in more detail in sections 2.1.1, 2.1.2 and 2.1.3, respectively.

2.1.1 User Intent

As mentioned previously, to perform program synthesis there must be a way for the user to express his intent. The user intent indicates the desired behavior of the program to be generated by the synthesizer.
Specification Given an input x and an output value y, a specification Ω is the description of the user's intent, such that Ω(x, y) is True if and only if y is the desired output value for x.
Despite all the progresses in program synthesis solutions, expressing user intent still remains a significant challenge. The first approaches on program synthesis, such as deductive synthesis, required the user intent to be expressed using a complete formal specification, which in most cases is harder than writing the program itself.
Using a complete specification might become as challenging as the underlying programming task.
However, when not specific enough, there might be more than one program that satisfies the provided specification and end up with a program that is not the desired one, due to the ambiguity of the specification.
The goal is to find an approach that allows finding the desired solution without the need for a very complex specification, i.e., find a balance between the completeness and ease of formulation of the specification.

2.1.2 Program Space

Once the specifications needed to express the user intent are provided, the synthesizer is now able to perform a search over the program space in order to find the desired program.

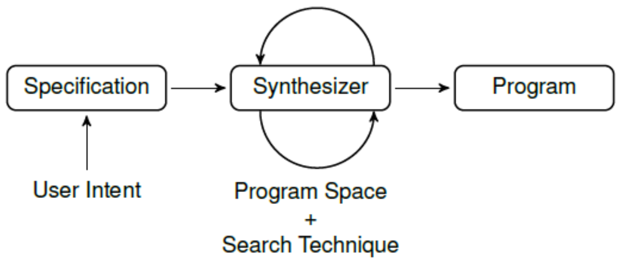

Figure 2.1: Program synthesis dimensions

Program Space is the space containing the set of all programs that can be written using a given programming language.
The program space for a given programming language is infinite, which leads to another challenge: the dimension of the program space. In order to tackle this challenge, one of the many possible approaches is to restrict the program space by imposing an upper bound on the number of lines or instructions that a program can have. However, the size of this restricted program space grows exponentially as the upper bound grows or as more components are added to the language.
A possible approach to reduce the restricted program space is making use of a pruning technique, such as domain-specific heuristics, restricting the program space using some program complexity metrics such as size or restrict the programs language using a Domain-Specific Language (DSL).
Domain-Specific Language defines both the syntax and the semantics of the language in which the synthesized programs are written, providing the appropriate notions and abstractions for a particular domain or problem.

2.1.3 Search Techniques

In order to find the intended program, one needs to search the program space for a program that satisfies the specification. The specification and the knowledge about the context of the problem are used in order to guide the search process. To do so, there are four main search techniques in program synthesis, from which we will be describing one in more detail in this section.
Enumerative Given a specification and a Domain Specific Language (DSL), the enumerative based approach consists in enumerating the programs that are in the search space using some heuristic to define the order in which they are enumerated, which can be program size, complexity, among others. Then, for each program, it checks if it satisfies or not the specification. In Figure 2.2 we can see an illustration of the enumeration process.
The enumerator is responsible for enumerating the candidate programs. These candidate programs are sent to a verifier which checks the consistency of the program according to the specification provided by the user. If the program is consistent, then it is returned to the user; otherwise, the enumerator must provide a new program to be verified. Although this sounds very simple, an

enumerative search approach may not scale up. Hence, it is important to have some pruning or a good ranking technique, in order to perform the search of the program space in a more efficient and effective way.

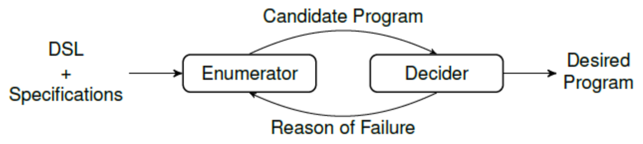

Figure 2.2 : Enumerative search program synthesis

3. RELATED WORK

In this chapter we discuss previous work related to this project. We focus in three main areas of program synthesis, namely inductive synthesis (3.1), with more emphasis on programming-by-example, program sketches (3.2) and enumeration-based program synthesis (3.3).

3.1 Inductive Synthesis

As described in section 2.1.1, expressing user intent might reveal to be a challenging task. Deductive synthesis approaches require the user intent to be provided as a complete formal specification, which in most cases is as demanding as writing the program itself. The process of generating a program from high-level formal specifications is called formal synthesis.

The need to make formal synthesis methods simpler led to the appearance of new inductive synthesis approaches based on inductive specifications such as input-output examples, like the FlashMeta framework for inductive program synthesis [23], which allows synthesizer developers to generate efficient synthesizers from a DSL definition.

3.1.1 Programming-by-Example

Programming-by-Example (PBE) is a sub-field of program synthesis that focuses on input-output example-based specifications. One of PBE's main goals is automating certain classes of programming tasks, which has proven extremely useful for end-users since it is easier to provide examples rather than a formal specification of the constraints, but also very useful to developers since it provides a tool for automating repetitive and tedious programming tasks in the form of informal specifications.

This approach is used in a wide range of domains, such as automating manipulations in spreadsheets like FlashFill [10], which allows users to quickly perform repetitive string manipulations in Excel by providing a very small set of examples of the expected behavior, without the need to write complex macros. Other examples include automating data preparation tasks [2, 7, 9, 17], regular expression synthesis [31], and SQL queries [30, 32].

Input-output examples enjoy a set of unique properties which sets PBE apart as a separate subfield of program synthesis. These properties are ease of use and ambiguity of the specification. As mentioned before, this approach provides the user a simpler and easier way to specify user intent for a given program, but they are also simpler to explain and verify, which is the reason why this is an ideal approach for users without programming background.

But, alongside the ease of use, comes the ambiguity of the provided solutions. PBE is highly dependent on the quality of the provided examples, increasing the likelihood of obtaining several programs that satisfy the input-output examples but do not accurately capture the user intent, which may lead to an increasing program space. Which leads us to some of the main program synthesis challenges: ambiguity resolution, since we do not want to just find any program that satisfies the input-output examples
but the intended one.

3.1.1 Ambiguity Resolution

One of the main characteristics in PBE, aside from the ease of use, is the ambiguity. Given a set of input-output examples, there might exist more than one program that is consistent with the examples, but does not satisfy the user intent, which is why examples are considered an under-specification. Therefore, it is important to establish a criteria for choosing a program from a given pool of candidates that satisfy the specification. To do so, two main solutions have been proposed [12]: Ranking [13, 23, 26] and Active Learning [11, 18].

Ranking Given a set of programs that are consistent with the examples, this approach performs a ranking of the programs according to their likelihood of corresponding to the user's intent and assigns a score to each one. In the end, the chosen programs correspond to the ones with the highest score.

Active Learning Is a common approach when the synthesizer finds more than one program that is consistent with the examples. Given two candidate programs, distinguishing inputs consist of using an input that produces a different output for each program, then ask the user which produced output is the correct one and discard the other program. Once the user selects the intended program the new input-output pair is added to the examples set. This technique is based on interaction with the user, in order to disambiguate between 2 candidate programs.

3.2 Program Sketches

One approach that has become very popular in program synthesis is the use of partial programs, also known as program sketches, to write code automatically [9, 20], data wrangling tasks [2], facilitate the use of software libraries [25], training neural networks [19] and solving component based synthesis problems [7, 24].

3.2.1 Sketch-based Program Synthesis

Solar-Lezama introduced an approach which allows the user to provide its specifications through a partial program referred to as sketch [27, 29].

A sketch expresses the high-level structure of an implementation but has holes which represent the low-level details. The key idea is to create an abstraction from the source code that clearly defines the semantics but not the syntax, this is, the sketch abstracts out names and

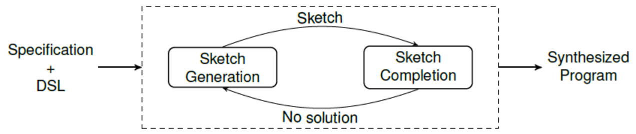

Figure 3.1: Sketch-based program synthesis

operations from a program, but keeps the program's structure, the order in which it executes methods, types of arguments and its return values. This approach is know as programming with sketches [28].

In program synthesis, we have already seen that the use of examples as a specification can be very useful. Among the various types of programming-by-examples approaches we have seen, sketches can be used to guide the structure of the intended implementation.

Also, it allows the user to focus on the algorithmic properties of the implementation rather than the low-level details. Solar-Lezama et al. [28] show that this approach improves the productivity and performance of programming tasks. The sketch-based synthesis process can be split in two stages: sketch generation and sketch completion, as illustrated in Figure 3.1. The first process consists in generating a sketch, using an automated sketch generation technique [8], in which the synthesizer enumerates the sketches according to some complexity metric, such as the sketch size, and a given DSL. Followed by the filling of the holes, with the use of a synthesizer to fill each hole with an according expression in order to generate a complete program, which corresponds to the sketch completion stage. This process is repeated until a valid solution is found according to the given specification.

3.3 Enumeration-based Program Synthesis

There exist several approaches to program synthesis, one of the most common being enumeration-based search. This technique consists of performing a search over the space of all candidate programs that can be generated from a given DSL [2, 9, 17, 21, 31]. The enumeration prioritizes programs according to some heuristic and returns the first program that satisfies the specification provided by the user. This technique is frequently used in many state-of-the-art synthesizers that also rely on logical deduction [2, 17, 22], where the space of candidate programs is encoded using either Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT).

As shown in Figure 2.2 the enumeration-based technique has two main components: an enumerator and a decider. The enumerator enumerates all the possible programs for a DSL given as input. For each enumerated program the decider will evaluate if it satisfies the specification provided by the user. For the particular case of PBE, this evaluation performed by the decider is done by executing the enumerated program using the input examples and checking if the output matches the corresponding output examples. If the output does not match the expected one we consider that program to be infeasible.

4. PURE FUNCTION SYNTHESIS

This chapter presents PUFS, a PBE-based **Pu**re **F**unction **S**ynthesizer, developed using an enumeration-based approach, both for enumerating programs as well as sketches. We start by providing a brief introduction to the OutSystems platform, as well as a description of the problem (Section 4.1), followed by the description of the enumeration-based sketch generation approach (Section 4.2) and the sketch completion approach (Section 4.3) as well as all the techniques used in its components.

4.1 Problem Formulation

This thesis was developed within the context of the OutSystems platform. OutSystems is a low-code development platform which provides a graphical user interface for the development of mobile and web applications, while allowing easy integration with other existing systems and the use of traditional textual programming (e.g. JavaScript, SQL) when needed. Its main goal is to enable an easier and faster development experience of enterprise-level applications.

In the OutSystems platform, business logic is defined using action flows. Pure functions are one type of action flow that produces an output given a set of inputs. These functions are characterized for having no side-effects and can be used within OutSystems expressions, which makes them useful for performing complex data transformations that are recurrent throughout the application.

An OutSystems expression is composed by operands and operators. The operands can be a literal (e.g. strings, numbers, Boolean values, etc.), any element available in the scope of the current expression, such as local variables, or function calls, or sub-expressions. The operators can be of type numeric, logic and Boolean, among others. However, in this work we are focusing on synthesizing pure functions that use built-in types such as Integer, Decimal, Text and Boolean, and built-in functions such as Math, Numeric and Text.

We are mainly focused in synthesizing pure functions, but for the scope of this thesis we are focused on the ones with just conditional expressions in form of If statements and assignment expressions without loops. The assignment expressions assign a value to a given variable. On the other hand, If statements consist of an expression to be evaluated in order to condition the control-flow of the function.

The goal of this thesis is to synthesize this type of functions using program synthesis, from an input-output example-based specification. Since pure functions return an output from a given set of inputs, these represent an appropriate candidate to apply this technique. These input-output examples represent the expected behavior of a flow.

In order to simplify the synthesis task sketches are used. We follow a two-stage approach consisting of sketch generation and sketch completion, further described.

The user provides an input file containing both the specification, in the form of input-output examples. The

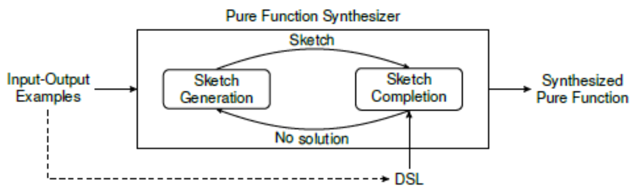

Figure 4.1: Pure Function Synthesizer.

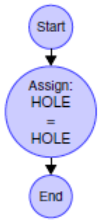

Figure 4.2: Partial flow example.

structure of the input file and assemble of the DSL is further described in section 4.3.
In order to simplify the synthesis task sketches are used. We follow a two-stage approach consisting of sketch generation and sketch completion, further described in section 4.2 and section 4.3 respectively. During the sketch generation phase we generate the partial flows to be completed with the during the sketch completion phase, in order to get a complete flow corresponding to a program. These partial flows consist of flows with holes in place of the expressions of each Assign and If node. A flow is considered correct if, once complete, it returns the expected output for the respective input for all the input-output pairs given as specification. If a valid solution is not found, i.e., it is not valid according to the input-output examples, then the synthesizer tries to find a solution using another sketch and so on.
The overview of the architecture of our framework is illustrated in figure 4.1.

4.2 Sketch Generation

The sketch generation consists in generating a sketch of a flow, i.e., a flow with holes in place of the assignment and If node expressions as illustrated in figure 4.2.
This generation process consists of a enumerative approach that enumerates several sketches up to a pre-specified size. The size of a flow corresponds to the number of nodes in that flow.
In our approach, graphs are used as a representation of the flows, since a flow is akin to the control-flow graph of an application. Given that, we consider a flow to be a graph. Also, the graphs allow us to have a representation that gathers all the necessary information to enumerate the sketches, such as the neighborhood of each node, which expression is associated to the node, among others. Therefore, in this section, when we refer to a flow, we refer to its structure as a graph. We consider the size of a flow to be the number of nodes of the corresponding graph. Figure 4.3 shows the sketches that would be generated for a fixed size of 5. Given the desired size of the flow, the enumerator

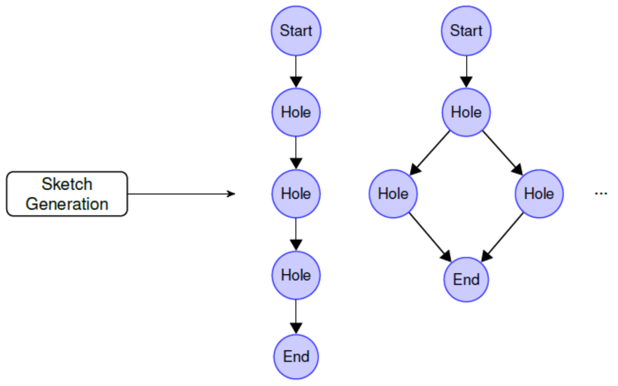

Figure 4.3: Enumeration of sketches given the desired size of the flow.

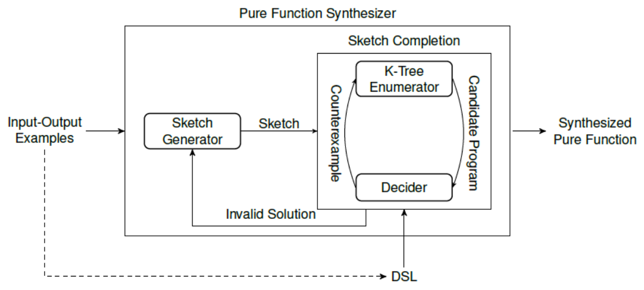

Figure 4.4: Overview of the PUFS synthesis process.

follows a recursive approach conditioned by the current size of the flow and the desired final size. We consider 5 possibilities, that correspond to when there are 0, 1, 2, 3 or more nodes away from reaching the desired flow size.
At each step of the recursion we check if there are If and non-If free nodes. Free nodes correspond to nodes which have no outgoing edges. We consider non-If free nodes to be nodes of type Start or Assign that have no outgoing edges and If free nodes to be If nodes with at most one outgoing edge or none, since If nodes must have 2 outgoing edges corresponding to the ”True” and ”False” edges.
At each new recursive step we check how many nodes are missing in the current graph in order to achieve the desired size. Then, based on the size of the current graph and the corresponding free nodes we add the possible nodes among Assign, If and End nodes. At the end of each step a new recursion begins using the updated graphs.

4.3 Sketch Completion

Once the sketch is generated, its holes must be filled in order to obtain a complete flow that corresponds to a correct and valid pure function. Within the sketch completion process we have two main components: the K-Tree Enumerator and the Decider. This process is illustrated in figure 4.4.

4.3.1 K-Trees Enumeration

Once the input file, with the input-output examples, along with the DSL are provided, the enumeration of the candidate programs takes place. The enumeration of the

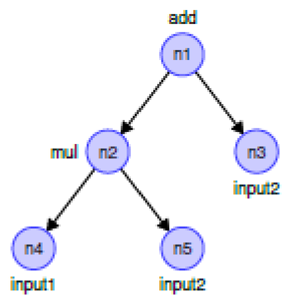

Figure 4.5: An example AST.

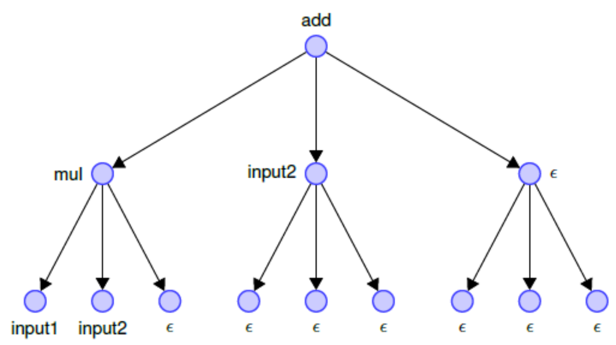

Figure 4.6: Example AST of figure as a k-tree.

candidate programs is guided by a sketch, i.e., the sketch is the flow's graph with holes instead of expressions.
In order to perform the enumeration of candidate programs, we need to use a structure that is capable of representing every possible program in the DSL. Programs are often represented using their Abstract Syntax Tree (AST) representation. An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of a program, where each internal node represents an operator, and the children represent the respective operands. For instance, the AST shown in figure 4.5 corresponds to the program *add(mul(input1; input2); input2).*
K-trees are a popular representation used in enumeration-based program synthesis due to its ability of representing every possible program for a given DSL. Therefore, k-trees are the representation used in PUFS. A k-tree is a tree of depth d, where every internal node has exactly k children and every leaf node is at depth d. For the current DSL supported by PUFS, the maximum arity among all DSL constructs is 3, meaning every k-tree will have 3 children, as shown in Figure 4.6.
In order to enumerate the possible programs using k-trees, the synthesizer encodes the trees as an SMT formula. A complete program can be extracted from a model of the SMT formula. A model that satisfies that formula represents the assignment of a symbol of the given DSL to each node in the trees. Within the possible tree encoding approaches, we have selected the line-based encoding. In the line-based encoding, a program is represented using a

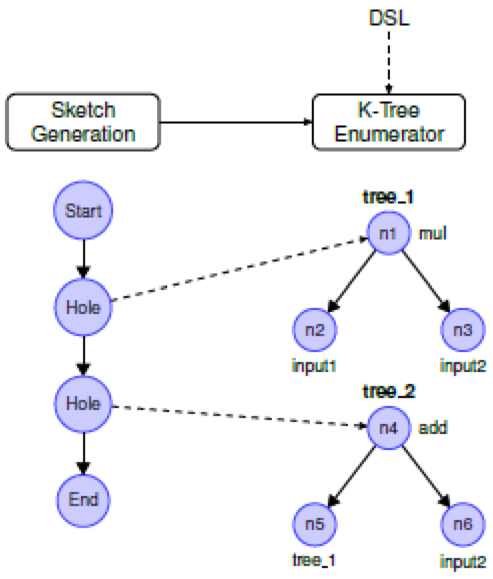

Figure 4.7: Enumeration of k-trees for a given sketch with two holes to fill.

sequence of trees of depth 1, where each tree represents one operation of the program, as in an imperative language.
PUFS uses an adaptation of the line-based encoding presented in SQUARES. A program representing a flow with Assign nodes only can be seen as a sequence of operations, therefore, we want to fill each hole using a k-tree rather than one single k-tree to represent the whole program, as shown in figure 4.7. However, when it comes to flows with If nodes that does not apply, and for that same reason we use an adaptation of the encoding instead of the original one. Furthermore, the trees are enumerated in increasing depth until a solution is found or until a timeout is reached.
In the following sections we describe the variables and constraints used to encode the line-based.

4.3.2 Line-based Encoding with Conditionals

When considering flows with Assign nodes only, the previous encoding, described in section 3.3, would be enough, since a program, with assignment expressions only, can be seen as program written in an imperative language where each line would be the expression associated to each Assign node. However, in the presence of If nodes, the structure of our programs is not so straight forward. Therefore, in order to synthesize flows with conditional expressions we have implemented the following constraints.
Recall that *D* is the *DSL*, *Prod(D)* the set of production rules in *D* and *Term(D)* the set of terminal symbols in *D*. Furthermore, *Types(D)* represents the set of types used in *D* and *Type(s)* the type of symbol *s ϵ Prod(D) U Term(D).* If *s ϵ Prod(D)*, then *Type(s)* corresponds to the return type of production rule *s*.
Consider Σ the set of symbols used in the program. Besides the production rules and terminal symbols, there is one

additional symbol ret for each line in the program. Let *Ret = {ret*$_i$ *: 1≤ i≤ n}* represent the set of return symbols in the program, then Σ = *Prod(D) U Term(D) U Ret*.
Furthermore, each symbol is assigned a unique positive identifier. Let *id : Σ⟶ N*$_0$ be a one-to-one mapping function that maps each symbol in Σ to a unique positive identifier and *tid* : *Types(D)* ⟶ *N*$_0$ be a one-to-one mapping function that maps each symbol type to a unique positive identifier. Finally, since some operations in the DSL have a smaller arity than *k*, the empty symbol ϵ is introduced, so that every leaf node has an assigned symbol. We assume *id(ϵ) = 0*.

Encoding variables. Consider a sketch with n holes to fill, where the maximum arity of the operators used in the expressions is k, and each hole will be filled using a line, we have the following variables:

- _ *O = {op*$_i$ *: 1 ≤ i ≤ n}* : each variable *op*$_i$ represents the production rule used in line *i*.
- _ *T = {t*$_i$ *: 1 ≤ i ≤ n}* : each variable *t*$_i$ represents the return type of the expression in line *i*.
- _ *A = {a*$_{ij}$ *: 1 ≤ i ≤ n, 1 ≤ j ≤ k}* :: each variable a$_{ij}$ represents the symbol corresponding to argument *j* in line *i*.

To ensure the enumerated programs are well-typed we need to add the following constraints.

Operations constraints. The operations in each line must be production rules.

$$\forall\, 1 \le i \le n : \bigvee_{p \in Prod(D)} op_i = id(p)$$

If a node *i* corresponds to an If node, then the line used to fill that node's hole must be a production rule for which the return type is Boolean. Let *BooleanProd(D)* be the set of such production rules that appear in the DSL *D*, and *HoleType(i)* the node type of hole *i*.

$$\forall\, 1 \le i \le n : HoleType(i) = If \implies \bigvee_{p \in BooleanProd(D)} op_i = id(p)$$

The return type of each line is the return type of its production rule.

$$\forall\, 1 \le i \le n,\ p \in Prod(D) : op_i = id(p) \implies t_i = tid(Type(p))$$

Given a sketch with more than one hole to fill, the arguments of an operation i used in a hole must be either terminal symbols or return symbols from previous holes.

Arguments. Given a sketch with more than one hole to fill, the arguments of an operation *i* used in a hole must be either terminal symbols or return symbols from previous holes.
However, if the sketch to be completed has If nodes, there will be more than a single execution path, so, the results of an operation can only be used within the following operations of the same execution branch. Therefore, we have the following constraint. Let *PreviousHoles(i)* be the set of lines used in previous holes from the same execution path as node *i*, excluding lines that are used to fill If nodes.

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

The arguments of an operation *i* must have the same types as the parameters of the production rule used in the operation. Let *Type(p, j)* be the type of parameter *j* of production rule *p*, where *p ϵ Prod(D).* If *j > arity(p)* then *T(p, j) = ϵ*. Hence, there are the following constraints when a return symbol is used as an argument of an operation:

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ 1 \le j \le arity(p),\ 1 \le r < i :$$

$$((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \implies (t_r = tid(Type(p, j)))$$

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

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ 1 \le j \le arity(p),$$

$$s \in \{r \in Term(D) : Type(r) \ne Type(p, j)\} :$$

$$(op_i = id(p)) \implies \neg(a_{ij} = id(s))$$

The arity of an operation *i* can be smaller than *k*, in that case, the empty symbol ϵ is assigned to the arguments above the productions arity.

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ arity(p) < j \le k :$$

$$(op_i = id(p)) \implies (a_{ij} = id(\epsilon))$$

Output. Let *Type(out)* be the type of the program's output and *P*$_{out}$ *ϵ Prod(D)* be the subset of production rules which return type equal to *Type(out)*, i.e., *P*$_{out}$ = *{p ϵ Prod(D) : Type(p) = Type(out)}*. Given that a flow can have multiple nodes pointing to an End node, there is more than one possible output result. Consider *L* the set of all lines corresponding to nodes that point to an End node. Since the last line of a program corresponds to the program's output, the operation of each one of the lines in *L* must be one the productions in *P*$_{out.}$

$$\forall\, l \in L : \bigvee_{p \in P_{out}} (op_l = id(p))$$

Input. Let *I* be the set of symbols provided as input by the user. Each input must be used at least once:

$$\forall s \in I : \bigvee_{1 \le i \le n} \bigvee_{1 \le j \le k} (a_{ij} = id(s))$$

Lines once or more times. We are interested in enumerating programs where the result of an operation can be used in the following operations 1 or more times. Hence, we have the following constraint.

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

4.3.1 Decider

Once the sketch is filled with the corresponding expressions in each node, the decider evaluates if the resulting program satisfies the specification. To perform this evaluation, we

Table 5.1: Comparison of number of instances solved for the different timeout values.

Timeout(s)	60	120
Without constants	72	125
With constants	121	132

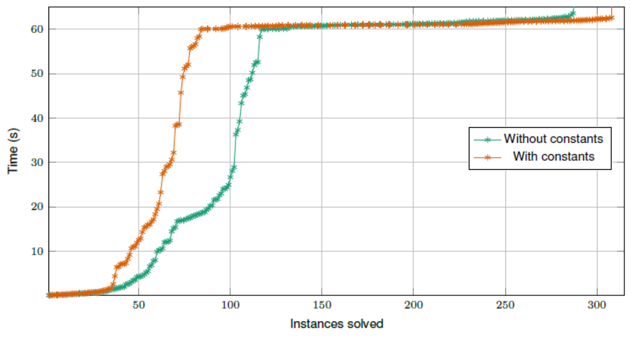

Figure 5.1: Number of instances solved throughout time for a timeout of 60 seconds.

developed a flow interpreter, which takes a graph that represents a flow and interprets the resulting program using the input values to obtain the corresponding outputs of that program. If, for every input, the program returns the corresponding output from the specification the decider considers that a solution was found. Otherwise, a new sketch is generated to be filled by the synthesizer. This process is repeated until a solution is found or until a time limit is reached.

5. EVALUATION

The goal is to evaluate how many pure functions PUFS is able to synthesize and how quickly. We are also interested in evaluating how the quality of the examples affects the performance of the synthesizer in terms of run time and program quality. Besides performing an evaluation based on properties of the examples, we also evaluate if providing additional information, such as constants to be used in the program, has a significant impact in the performance.

We developed PUFS on top of the TRINITY [17] synthesis framework. The synthesizer is implemented in Python 3.6 and it uses the Z3 SMT solver [4] with theory of Linear Integer Arithmetic to solve the SMT formulas generated during the synthesis process. The results presented in this section ere obtained using an Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz, with 16GB of RAM, running Ubuntu 18.04 LTS, with time limit of 300 seconds.

5.1 Experimental Results

We want to evaluate the impact of the number and quality of the input-output examples on the performance of our synthesizer, in terms of runtime and program quality. Additionally, we evaluate the impact of providing constants

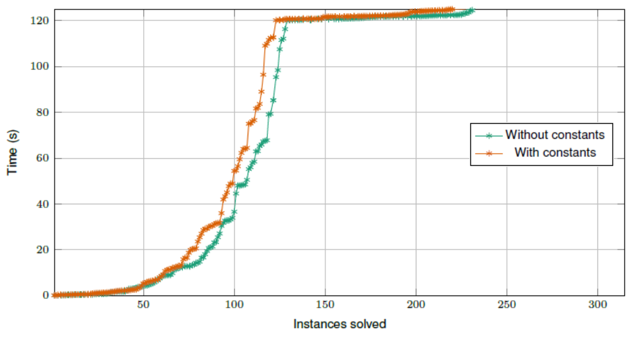

Figure 5.2: Number of instances solved throughout time for a timeout of 120 seconds.

along with the input-output examples. For each instance we ran the synthesizer with 5 input-output examples.

For instances in which the solution is expected to use constants we provided the corresponding constants in the specification file along with the input-output examples. These constants consist of integer, decimal and string values. We ran the synthesizer for instances with constants and with no constants provided. This is due to the fact that the synthesizer is not able to synthesize programs with constants for now, so solutions that require constants would not be found using the present configuration.

Table 5.1 shows the results of both approaches in terms of the number of instances solved and limit time considered.

We consider an instance solved it the synthesized program satisfies the input-output examples. However, even though it is considered solved it does not mean it is correct, i.e. it might not match the intended solution. Matching the user intent means that the solution must satisfy the specification as well as capturing the user intent (this verification is performed manually).

5.2 Discussion

From the presented results, we can verify that as the complexity of the program increases so does the time to find solution. As the enumeration process is done in an ascending order of size of the sketch, the bigger the sketch we want to complete the longer it will take to complete it and find a solution that satisfies the specifications. We need to take into account that the bigger the size of the program we are synthesizing, the longer it will take for the sketch completion stage to be completed, due to the fact that the number of holes we wish to fill increases the number of expressions to be synthesized.

However, from the comparison between the different timeout limits and the number of instances that the synthesizer was able to solve during each of those time intervals, we can verify that the difference in the time given does not have a significant impact on the number of instances solved, especially when using constants.

Regarding the use of constants, from plot 5.1 and table 5.1, we can verify that the use of constants has a significant impact in the number of solved instances, which is more evident in instances solved under 1 minute. This impact would be expected due to the fact that the synthesizer does not enumerate candidate programs containing constants, unless these are provided by the user. However, the addition of constants also increases the time it takes to find a solution, due to the fact that more candidate programs are enumerated. Which would explain the results for the timeout of 2 minutes, illustrated in 5.2, where we verify that the number of solved instances without constants is almost the same as when providing the constants. For this particular case, it is important to evaluate if the trade-off between the additional time spent enumerating more programs, compensates by finding a solution in a reasonable time.

Another aspect to have in consideration is the number of solutions that do not correspond to the intended one. The use of constants, especially when used in conditional expressions to limit a range of values, can lead to solutions that do not match the expected one, if the provided input-output examples do not cover the limits of that range.

6. CONCLUSIONS & FUTURE WORK

In this thesis, we tackle the problem of synthesizing pure functions from examples in the OutSystems platform. We focus on functions that manipulate integers, decimals, text and Booleans. The OutSystems platform main goal is to provide an easier and faster experience in development and integration of web and mobile applications. This platform gives users, with no programming background, a tool that allows them to develop an application with no need for specific knowledge, and also provides a faster and automated approach to users with more specialized knowledge. Therefore, it is in our interest to provide a simple approach for the user to be able to generate the pure functions with only a small number of examples in just one click. These types of functions come across very often, in the form of data wrangling tasks, among others. Having such a repetitive type of task might become tedious and add more complexity to the overall tasks, which leads to a need of automating this type of functions. In this dissertation we presented a novel approach to synthesize pure functions in the form of flows in the OutSystems platform, from a set of input-output examples. We survey the state of the art in program synthesis and implemented PUFS, a PBE-based pure function synthesizer. The synthesizer employs the use of sketches as the underlying structure of our programs and enumerative search, where SMT is used to search the program space. We tested PUFS in a set of real-world examples of pure functions developed in the OutSystems platform, from which the results of our experiments revealed we are able to synthesize 33% of the benchmarks within less than a minute. However, the results also revealed that in a significant number of examples we were not able to find a solution within a limited amount of time
due to the dimension of the program space or due to an incomplete specification. Given the experimental results, we believe the current solution could benefit from some pruning techniques, in order to reduce the search space and possibly overcome the time limitation to find a solution. It would also be interesting to use a ranking technique as an ambiguity resolution technique in order to guide the synthesizer into finding a program that is more likely to lead to a solution that satisfies both the specification and corresponds to the user intent. Moreover, it is important to explore if this approach scales if the DSL was to be extended, given that it would be interesting to extend the current DSL to support more data types and more functions. Besides the DSL it would be particularly interesting to support other type of nodes that allow other types of operations.

REFERENCES

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, 2017. URL https://openreview.net/ forum?id=ByldLrqlx.

[2] Y. Chen, R. Martins, and Y. Feng. Maximal multi-layer specification synthesis. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, 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. ACM, 2019. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338951. URL https://dl.acm.org/citation.cfm?id=3338906.

[3] A. Cohen and M. T. Vechev, editors. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.

[4] 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. doi: 10.1007/978-3-540-78800-3n 24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[5] 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. A. 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. doi: 10.1145/2884781.2884786.URL https://doi.org/10.1145/2884781.2884786.

[6] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. In D. Precup and Y. W. Teh,

editors, Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017, volume 70 of Proceedings of Machine Learning Research, pages 990–998. PMLR, 2017. URL http://proceedings.mlr.press/v70/devlin17a.html.
[7] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In Cohen and Vechev [3], pages 422–436. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062351.
[8] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In G. Castagna and A. D. Gordon, editors, Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 599–612. ACM, 2017. ISBN 978-1-4503-4660-3. URL http://dl.acm.org/citation.cfm?id=3009851.
[9] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. 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 420–435. ACM, 2018. doi: 10.1145/3192366.3192382.
[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, 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. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL http://dl.acm.org/citation.cfm?id=1926385.
[11] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, editors, 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012, pages 8–14. IEEE Computer Society, 2012. doi:10.1109/SYNASC.2012.69. URL https://doi.org/10.1109/SYNASC.2012.69.
[12] S. Gulwani. Programming by examples: applications, algorithms, and ambiguity resolution. In W. Vanhoof and B. Pientka, editors, Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017, page 2. ACM, 2017. ISBN 978-1-4503-5291-8. doi: 10.1145/3131851.3131853. URL https://doi.org/10.1145/3131851.3131853.
[13] S. Gulwani and P. Jain. Programming by examples: PL meets ML. In B. E. Chang, editor, Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings, volume 10695 of Lecture Notes in Computer Science, pages 3–20. Springer, 2017. doi: 10.1007/978-3-319-71237-6n 1. URL https://doi.org/10.1007/978-3-319-71237-6_1.
[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In M. W. Hall and D. A. Padua, editors, 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. ACM, 2011. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506.
[15] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. Foundations and Trends in Programming Languages, 4(1-2):1–119, 2017. doi: 10.1561/2500000010.
[16] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, 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. ACM, 2010. ISBN 978-1-60558-719-6.
[17] 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. doi: 10.14778/3352063.3352098. URL http://www.vldb.org/pvldb/vol12/p1914-martins.pdf.
[18] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In C. Latulipe, B. Hartmann, and T. Grossman, editors, Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015, pages 291–301. ACM, 2015. doi: 10.1145/2807442.2807459. URL https://doi.org/10.1145/2807442.2807459.
[19] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine. Neural sketch learning for conditional program generation. In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. URL https://openreview.net/forum?id=HkfXMz-Ab.
[20] M. I. Nye, L. B. Hewitt, J. B. Tenenbaum, and A. Solar-Lezama. Learning to infer program sketches. In K. Chaudhuri and R. Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, volume 97 of Proceedings of Machine Learning Research, pages 4861–4870. PMLR, 2019. URL http://proceedings.mlr.press/v97/nye19a.html.
[21] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In T. Schiex and S. de Givry, editors, Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings, volume 11802 of Lecture Notes in Computer Science, pages 583–599. Springer, 2019. doi: 10.1007/978-3-030-30048-7n 34. URL https://doi.org/10.1007/978-3-030-30048-7_34.

[22] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. SQUARES : A SQL synthesizer using query reverse engineering. Proc. VLDB Endow., 13(12):2853–2856, 2020. URL http://www.vldb.org/pvldb/vol13/p2853-orvalho.pdf.
[23] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In J. Aldrich and P. Eugster, editors, 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. ACM, 2015. ISBN 978-1-4503-3689-5. URL http://dl.acm.org/citation.cfm?id=2814270.
[24] K. Shi, J. Steinhardt, and P. Liang. Frangel: Component-based synthesis with control structures. CoRR, abs/1811.05175, 2018. URL http://arxiv.org/abs/1811.05175.
[25] K. Shi, J. Steinhardt, and P. Liang. Frangel: component-based synthesis with control structures. PACMPL, 3(POPL):73:1–73:29, 2019.
[26] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In D. Kroening and C. S. Pasareanu, editors, Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, volume 9206 of Lecture Notes in Computer Science, pages 398–414. Springer, 2015. ISBN 978-3-319-21689-8.
[27] A. Solar-Lezama. Program Synthesis by Sketching. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.
[28] A. Solar-Lezama, R. M. Rabbah, R. Bod´ık, and K. Ebcioglu. Programming by sketching for bitstreaming programs. In V. Sarkar and M. W. Hall, editors, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, pages 281–294. ACM, 2005. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065045.
[29] A. Solar-Lezama, L. Tancau, R. Bod´ık, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, 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. ACM, 2006. ISBN 1-59593-451-0.
[30] C. Wang, A. Cheung, and R. Bod´ık. Synthesizing highly expressive SQL queries from input-output examples. In Cohen and Vechev [3], pages 452–466. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.
[31] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett. Sketch-driven regular expression generation from natural language and examples. Trans. Assoc. Comput. Linguistics, 8:679–694, 2020. URL https://transacl.org/ojs/index.php/tacl/article/view/2135.
[32] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In E. Denney, T. Bultan, and A. Zeller, editors, 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pages 224–234. IEEE, 2013. URL https://ieeexplore.ieee.org/xpl/conhome/6684409/proceeding.