Next-Gen Pure Function Synthesis

Joana Maria Leal Coutinho
joana.coutinho@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Portugal

ABSTRACT

OutSystems is a low-code platform that allows users to create their applications through graphical interfaces instead of hand-coded computer programming. However, in the OutSystems platform, business logic is implemented through action flows, a graph that illustrates the intended logic, which requires the user to think like a traditional developer when implementing such flows leaving one desiring to automate it.

In this work, we seek to extend previous work of automating logical flows in the OutSystems platform, increasing the performance and allowing more complex operations and domains. More specifically, the goal is to add support for synthesizing list manipulations and data aggregation on the OutSystems platform. The solution focuses on pure function synthesizing using programming by example as the specification method and the search technique is a combination of sketch enumeration and satisfiability modulo theories.

1 INTRODUCTION

Nowadays, more and more people have access to technology devices, such as smartphones or computers. However, the learning curve needed for a person to program such devices is significant. OutSystems is a software automation platform that allows users to create their applications through graphical interfaces instead of traditional text-based programming. The goal of OutSystems is to provide efficient tools that are easy to use and responsive in just a few seconds, not requiring the user to acquire new skills. However, in the OutSystems platform, business logic is implemented through action flows, a graph that illustrates the intended logic, which requires the user to think like a traditional developer when implementing such flows leaving one desiring to automate it.

Program synthesis consists of automating the creation of a program according to a certain specification. Program synthesis enables one to build computer programs without any knowledge of programming, by shifting the effort from writing an implementation to providing a specification of the intended semantics instead. Hence, program synthesis seems like a good form of automating the implementation of action flows used in the OutSystems platform.

A pure function is a function that always returns the same value for the same input and produces no side effects, such as the modification of global variables or databases. For program synthesis, pure functions can simplify the reasoning process significantly by removing the need to reason about side effect. This scenario fits naturally into the programming-by-example paradigm because pure functions allows us to be confident the output is consistent.

In this work, we seek to extend previous work by creating a new generation of pure function synthesizers that support more complex scenarios and have a more efficient performance. More specifically, the goal is to add support for synthesizing list manipulations and data aggregation on the OutSystems platform. To the best of our knowledge, this is the first work that integrates this kind of operations into a single framework targeting action flow synthesis, taking us one step closer to a fully declarative development experience.

Motivation Example. Suppose there is a director of a faculty who wants to present a list of the working personnel. The director wants a function that, by default, returns a list of the professors. However, when the function receives a Boolean `include_support_staff` as True, the function should also return the remaining personnel, such as the human resources department. If we decompose this problem, assuming there is a database of professors and one for support staff, we can see that we want to, depending on the value of `include_support_staff`, either obtain only the professors, or obtain both the professors and support staff joining them into a single list.

One of the goals of OutSystems is to allow citizens to, without any knowledge of programming or SQL querying, develop enterprise-grade applications. The implementation of this logic in OutSystems might not be easy for such a user, given that this problem requires the knowledge of SQL querying and the logic of the OutSystem's platform. Instead, our framework allows the user to just provide a specification composed of input/output examples, which is more natural for the user.

For this problem, the director would need to provide at least two examples: the case where the argument `include_support_staff` is True were the two tables from the input are joined and returned in a list; and then the case when the value is False where a list with only the table of professors is returned.

Contributions. In this thesis, we propose PUFS-X, a framework for synthesizing action flows with assignment, conditional, list manipulation and data aggregation operations. We build upon previous work on pure function synthesis, the PUFS framework. The main contributions are as follows:

- Several performance improvements to the PUFS framework creating PUFS+, such as:
  - pruning of redundant or invalid sketches and programs by considering symmetries in the action flows and more fine-grained type information;
  - efficient modelling of constants;
  - rarity threshold to reduce the operations of the synthesizer;
- Creation of the PUFS-L framework, which adds list manipulation capabilities to the PUFS+ framework;
- Creation of the PUFS-SQL, which adds data aggregation capabilities to the PUFS+ framework;
- Creation of the PUFS-X, which joins all features into a single synthesizer.

© 2021 This work was supported by OutSystems, by national funds through *Fundação para a Ciência e Tecnologia*, with reference BI|2020|82, and project GOLEM 45917 funded by FEDER and FCT.

2 FUNDAMENTAL CONCEPTS

This section provides the fundamental concepts necessary to understand the remaining of the document.

2.1 Program Synthesis

Definition 2.1 (Program Synthesis). Program synthesis consists of automatically deriving a program from a specification through search techniques and a defined program space.

The Program Synthesis process consists of choosing a method for the user specification, defining a program space, and a search technique.

Definition 2.2 (Specification). Given an input $x = (x_1, x_2, ..., x_n)$ and output y, a formula ϕ is a specification such that $\phi(x, y)$ is True, if and only if y is the desired output of x.

There exist multiple types of user intent specifications, ranging from formal specifications, such as formulations, to more informal ones such as input-output examples or natural language. An informal specification is considered more intuitive for user, whereas a formal specification requires knowledge of mathematics and formulation for the user, which can prove to be as hard as writing the program itself. Examples of the latter approach are the first innovative papers in the late 60s [4], and early 70s [6]. In this work, we use the programming by example method, which relies on an input-output example based specification.

Example 2.3. An input-output example specification can be the input (1, 2, 3, 4) with the corresponding output (2, 4, 6, 8). A program that satisfies this specification would receive an input and multiply it by two.

A challenge of an informal approach is finding the perfect balance between completeness and simplicity for the specification. If too specific, the synthesizer may take a much more time to create the program than needed. However, if too broad, the synthesizer might return a program that satisfies the specification but not the user's true intentions.

Program Synthesis is an undecidable problem, one for which it is impossible to find an algorithm that can always give the correct answer. Hence, a search needs to be performed in the program space to find a program that satisfies the user's intent.

Definition 2.4 (Program space). A program space is the set of all programs that can be written using a given defined language.

The program space grows exponentially with the number of possible candidates and their corresponding size. Thus, if we search every possible combination, there are neither guarantees of efficiency nor guarantees of termination of the search. To minimize the program space's size, instead of using full-featured programming languages such as Python, domain specific languages are used.

Definition 2.5 (Domain Specific Language). A Domain Specific Language (DSL) is a language for a specialized domain, with restrictions that simplify the program space.

Example 2.6. A simple DSL of operations over lists, where N is the start symbol, is specified below. This DSL allows us to synthesize programs that use operations such as the filtering or sorting of lists. Suppose we want to synthesize a program that only performs list manipulations. In that case, we could significantly increase a synthesizer's performance by providing this DSL instead of a full-featured language.

$$N \rightarrow \texttt{0} \mid \ldots \mid \texttt{9} \mid \texttt{head}(L) \mid \texttt{last}(L) \mid \texttt{sum}(L) \mid \texttt{max}(L) \mid \texttt{min}(L)$$

$$L \rightarrow \texttt{get}(L, N) \mid \texttt{sort}(L) \mid \texttt{filter}(L, F)$$

$$F \rightarrow \texttt{geq} \mid \texttt{leq} \mid \texttt{eq}$$

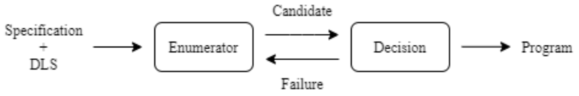

Figure 1: Enumerative search process

There are multiple search techniques that can be pursued, given a user specification and a program space. In this work, we use a combination of sketch enumeration and satisfiability module theories.

Enumerative search is the most common technique and consists of ordering the program space according to a heuristic, followed by iterating through it to find a program that matches the specification. Figure 1 illustrates the enumerative search process. The enumerator step chooses a candidate program, and the decision step verifies whether the candidate satisfies the user's intent. The process repeats until a satisfiable program is found.

Examples of successful enumerative search algorithms are Unagi [2], an Offline Exhaustive Enumeration over the DSL program space, or the synthesizing of geometry constructions [5].

2.2 The Sketching Approach

Automatically creating a program combines high-level insight about the problem and low-level implementation details. The latter comes naturally to computers. However, the former is much easier for a human than for a computer. Thus, Solar-Lezama introduced the concept of sketching [10, 11], a form of program synthesis that allows programmers to specify their high-level insight about a program, leaving the computer to determine the low-level details.

Definition 2.7 (Sketch). A sketch or a partial program is a program with holes.

2.3 Satisfiability Modulo Theories

Definition 2.8 (Satisfiability Modulo Theories). The Satisfiability Modulo Theories (SMT) problem is a generalization of Boolean satisfiability (SAT). Solvers that use SMT check the satisfiability of first-order logic formulas with use of theories such as theory of real numbers, theory of integer arithmetic, theory of strings. Given a theory T, a T-atom is a ground atomic formula in T. A T-literal is either a T-atom t or its complement $\neg t$. A T-formula is composed of T-literals. Given a T-formula ϕ, the SMT problem decides whether a solution exists such that ϕ is satisfied.

Example 2.9. Consider that $\mathcal{T}$ is the Linear Integer Arithmetic (LIA) theory. $\phi = (x+y > 2) \land (x > 4) \land (y < 1)$, is an example of an SMT formula in LIA, where x and y are integers. We can see that ϕ is satisfiable and a possible solution would be $x = 5, y = 0$.

3 RELATED WORK

This section describes the first attempt at a pure function synthesizer for the OutSystems platform and SQL synthesizers.

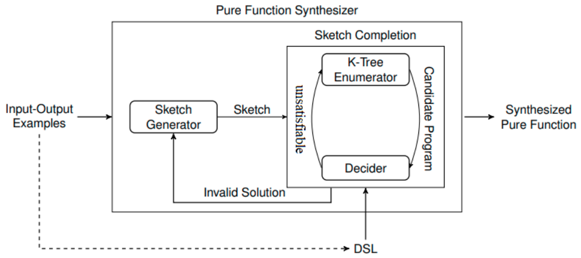

Figure 2: PUFS framework

3.1 PUFS Framework

Catarina Coelho proposed the first attempt at a pure function synthesizer for the OutSystems platform in her MSc thesis [1], the PUFS framework. The framework represents a program using a graph where a node can be an *Assign* node, which assigns a value to a given variable, or an *If* node which, according to a Boolean condition, allows two different paths depending on whether the condition is true or false. The usage of graphs as a method of representation parallels the representation used in the OutSystems platform.

The first step in the PUFS framework is the user specification, which is a set of input-out examples and a set of constants. The latter is used to guide the synthesizer to a more efficient search. The DSL used in the PUFS framework is composed of operands and operators provided by the OutSystems expression language. The operands can be literals (such as strings, numbers or Booleans), local variables, built-in functions or subexpressions. The operators are unary or binary such as +, − or =.

We must note that, due to pure function synthesizing, the DSL is constrained to operators that are considered pure, i.e., for the same inputs, the output is always the same not producing side effects such as changes to databases or global variables.

Figure 2 represents the architecture of the framework. As we can observe, we have two main steps: sketch generation and sketch completion. The main idea is that a candidate sketch is generated in the first step and then is completed in the second step if possible. Otherwise, a new candidate sketch is created, repeating the process. The sketch generator enumerates through partial flows, i.e., flows composed of *Assign* and *If* nodes such that its assignment expressions and condition expressions are holes to be filled.

The sketch completion step is where the holes of a sketch are filled. A k-tree is a recurrent tree representation used in enumeration-based program synthesis because of its ability to represent every possible program for a given DSL, where k is the largest arity among the operators. The k-tree enumerator enumerates through several trees, where each tree represents an expression that fills each hole. The PUFS framework encodes the tree as an SMT formula in order to obtain a concrete program by assigning a symbol of the DSL to each node.

When a sketch is completed, the decider checks if the respective candidate program satisfies the user's specification by comparing the output of the program ran on the input examples with the expected outputs. If the candidate does not satisfy, it returns to the k-tree enumerator to obtain a new candidate.

3.2 SQL Synthesizers

With the intent of integrating SQL queries to our work, in this sub-section we present two different SQL synthesizers.

SQUARES [7] is a PBE synthesizer for SQL queries and, besides the input/output examples, uses extra information from the user to improve the performance of the synthesizer, which includes a list of aggregation functions, a list of constants and the column names that can be used as arguments. SQUARES uses a DSL to specify the space of possible programs, which correspond to operations available in the libraries *dplyr* and *tidyverse* of the R programming language [1] that allow data-manipulation. SQUARES performs an enumerative search until either a solution is found or a the time limit is reached. Then, if a solution is found, the R program is transformed into a usable SQL query and returned to the user.

CUBES [3] was built upon the SQUARES framework and is recognized for the addition of new operations and the speed-up of the synthesis process by making use of multi-core processing.

4 NEXT-GEN PURE FUNCTION SYNTHESIS

In this section we propose the solution. We start by creating an improved version of the work done in pure function synthesis, the PUFS+ framework. We then extend the framework in two distinct manners: the addition of list manipulation capabilities, creating the PUFS-L framework; and the addition of data aggregation capabilities, creating the PUFS-SQL framework. Finally, the PUFS-X framework was created by joining all features into a single synthesizer.

4.1 PUFS+ Framework

The initial PUFS framework contains two types of nodes: the `Assign` node, which performs an assignment, and the `If` node, which, depending on a given condition, allows the execution of a program to follow one of two paths. Several different potential improvements were identified and implemented. We refer to the improved version of PUFS as PUFS+. In the following subsections, the major changes are presented.

4.1.1 Fine-grained DSL Types.

PUFS uses a single type in its DSL named `BuiltInType`.The usage of a single type allows the synthesizer to attempt operations that are not allowed by the synthesizer language, such as summing an Integer with a Boolean, thus resulting in the generation of many more invalid programs that have to be rejected by the decider. PUFS+ introduces 3 types to the DSL: `Numeric`,`Text` and `Boolean`.The type `Numeric` represents all numbers from integers to decimals. The `Text` type refers to any string. Finally, the type `Boolean` refers to `True` or `False`.All of the operators in PUFS were changed to their respective types, such as the operation *not* which changed from having the input and output as a value of type `BuiltInType` to type `Boolean`.

4.1.2 Node Connectivity Constraint.

PUFS allows nodes of a sketch to not be connected in their operators, which permits cases where a node performs an operation that is never used. In Figure 3, we can observe an example where the first node's operation is redundant since the following node does not use it as an argument. The symbol ϵ is used to represent an empty node. Note that, in order for the synthesizer to consider sketches with 2 nodes, all single node sketches must have already been

---

[1] The MSc thesis is awaiting publication

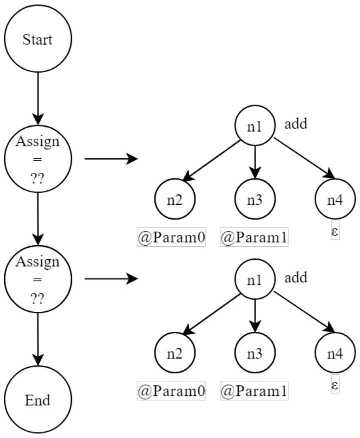

Figure 3: Lack of connectivity between nodes in the PUFS framework

exhausted. Thus, when multiple nodes are used, allowing unused nodes results in the generation of redundant programs.

PUFS+ ensures the connectivity between nodes of a sketch by adding a constraint to the SMT solver. This change forbids solutions such as the one seen in Figure 3, thus reducing significantly the number of possible attempts the synthesizer performs before finding the correct program. There are two possible encodings to ensure the connectivity of nodes: the Multi-Gen encoding and the Single-Gen encoding. The former allows a node to use any of the previous nodes whereas the latter only allows a node to use the immediate previous node. Thus, the Single-Gen encoding should increase the performance of the synthesizer when only the immediate previous node is required, because the search space reduces with the removal of solutions that use multiple previous nodes. However, it removes some possible solutions that would use more than one of the previous nodes at once, resulting in a trade-off between performance and completeness.

4.1.3 *Constants as Inputs.* PUFS requires an extra node for each constant used to transform it from the type `Const` to the usable type `BuiltInType`. Thus, given N constants, N extra nodes must be added to each sketch and then K nodes for the actual operators.

PUFS+ no longer considers a constant to be of type `Const` and simply models it as an extra input in the input-output examples. With this change, the extra constant nodes are no longer required. Thus, in contrast to the PUFS framework that requires $N + K$ nodes, the new framework only requires K nodes for the same solutions.

4.1.4 *Pruning of Redundant Sketches and Operators.* PUFS+ removes sketches whose final nodes of a sketch are `If` nodes, because the pure function property requires all of the flows to return an output. An `If` node, depending on a given condition, allows the execution of a program to follow one of two paths and only `Assign` nodes effectively return an output.

PUFS+ removes redundant operators from the DSL as follows:

- The operators *Lesser Than* and *Lesser or Equal Than* can be implemented using the operators *Greater Than* and *Greater or Equal Than*, respectively, by simply swapping the left and right-hand sides. A total of 12 operators were removed.
- *Equal* and *Different* with the new DSL were both duplicated to have 6 operators each for the different type combinations. However, the operators `eq_text_boolean` and `eq_boolean_text` are equivalent. The same occurs with the comparison of types `Text` and types `Numeric`. Thus, 4 operators in total can be removed from the DSL, 2 variants of Equal and 2 variants of Different.
- Adding two values of type Text (`add_text_text`) is equivalent to the concatenation operator (*Concat*). Thus, we remove the operator `add_text_text`.

4.1.5 *Rarity threshold.* Some of the operators in the DSL are used more frequently than others. For example, operators such as *add* or *mul* are significantly more frequent than operators such as *sqrt* or *power*. Therefore, a new configuration parameter was implemented in the synthesizer that allows one to ignore sets of operators based on rarity.

SMT constraints

Now we will describe how the SMT line-based encoding [8] of the PUFS+ frameworks was adapted from the work done by Orvalho et. al. [9] and Catarina Coelho, where each line of the encoding is considered a node of a sketch.

The encoding represents a program as a graph of nodes where each node uses an operator from the DSL. Each node is represented using a k-tree of depth one, where k represents the largest arity among the DSL operators, which can use as arguments any of the inputs or the result of operators used in previous nodes.

4.1.6 *Encoding Variables.* Let D be the DSL. The set of production rules $Prod(D)$ in D consists of the production $AssignProd(D)$, i.e., $Prod(D) = AssignProd(D)$. The productions of a node correspond to the operators allowed in its type. Furthermore, $BooleanProd(D)$ denotes the set of productions that return a Boolean value. Besides the productions, we use $Term(D)$ to denote 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 $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ corresponds to the return type of production rule s.

Consider a program with n nodes, where the maximum arity of the operators used in the expressions is k. We have the following variables:

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

Let Σ denote the set of all symbols that may appear in the program. Besides the production rules and terminal symbols, we introduce one additional symbol *ret* for each node in the program. Let $Ret = \{ret_i : 1 \leq i \leq n\}$ represent the set of return symbols in the program, then $\Sigma = Prod(D) \cup Term(D) \cup Ret$. The usage of the *ret* symbol is necessary to represent the use of previous nodes in a sketch, i.e., a node may use as an argument of an operator the returning value of a previous node.

Each symbol is assigned a unique positive identifier. Let $id : \Sigma \longrightarrow \mathbb{N}_0$ be a one-to-one mapping function that maps each symbol in Σ to a unique positive identifier and $tid : Types(D) \longrightarrow \mathbb{N}_0$ be a one-to-one mapping function that maps each symbol type to a unique positive identifier. Finally, since some operators in the DSL have arity smaller than k, and hence will never use all k leaves, the empty symbol ϵ is introduced so that every leaf node has an assigned symbol. For instance, the operator *not* uses a single argument, thus, the remaining $k-1$ leaves are assigned the symbol ϵ. We assume $id(\epsilon)=0$.

There exists a configuration parameter that influences the SMT constraints, `use_single_gen`. If it is True, then the synthesizer uses the Single-Gen encoding and, if otherwise False, then the synthesizer uses the Multi-Gen encoding. Let $PreviousHoles(i)$ be a set of nodes. In the Multi-gen encoding, $PreviousHoles(i)$ is the set of nodes that contain all previous holes from the same execution path as node i, ignoring `If` nodes. In contrast, in the Single-Gen encoding, $PreviousHoles(i)$ is only the last previous node of i also ignoring `If` nodes.

4.1.7 *Constraints.*

The SMT constraints that encode the problem are as follows.

Operations. The symbol of each node must be a production rule.

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

Let $HoleType(i)$ be the node type of hole i. If a node i corresponds to an `If` node, then the node's hole must be a production with a Boolean return type.

$$\forall\, 1 \le i \le n: HoleType(i) = If \implies \bigvee_{p \in BooleanProd(D)} op_i = id(p) \tag{2}$$

If a node i corresponds to an `Assign` node, then the respective symbol must be a production in $AssignProd(D)$. For all i between 1 and n:

$$HoleType(i) = Assign \implies \bigvee_{p \in AssignProd(D)} op_i = id(p) \tag{3}$$

The return type of each node is the same as the return type of its production rule.

$$\forall\, 1 \le i \le n,\ p \in Prod(D): (op_i = id(p)) \implies (t_i = tid(Type(p))) \tag{4}$$

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

$$\forall\, 1 \le i \le n, r \in PreviousHoles(i),\ 1 \le j \le k: \bigvee_{s \in Term(D)\, \cup\, ret_r : r<i} a_{ij} = id(s) \tag{5}$$

The arguments of an operator i must have the same types as the respective parameters of the production rule used in the operator. Let $Type(p,j)$ be the type of parameter j of production rule p, where $p \in Prod(D)$. If $j > arity(p)$ then $T(p,j)=\epsilon$.

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ 1 \le j \le arity(p),\ 1 \le r < i: ((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \implies (t_r = tid(Type(p,j))) \tag{6}$$

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

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ 1 \le j \le arity(p),\ s \in \{r \in Term(D) : Type(r) \ne Type(p,j)\}: (op_i = id(p)) \implies \neg(a_{ij} = id(s)) \tag{7}$$

The arity of an operator i can be smaller than k; in that case, the empty symbol is assigned to the arguments that exceed the production's arity:

$$\forall\, 1 \le i \le n,\ p \in Prod(D),\ arity(p) < j \le k: (op_i = id(p)) \implies (a_{ij} = id(\epsilon)) \tag{8}$$

Output. Let $Type(out)$ be the type of the program's output, $P_{out} \subseteq Prod(D)$ be the subset of production rules with return type equal to $Type(out)$, i.e., $P_{out} = \{p \in Prod(D) : Type(p) = Type(out)\}$, $Ret = \{ret_i : 1 \le i \le n\}$ represent the set of return symbols in the program and End be the type of node that ends a flow and returns the effective output. Given that a flow can have multiple nodes pointing to an End node, there is more than one possible output result. Let L denote the set of all nodes that point to an End node. Since the last nodes of a program correspond to the program's output, the operator of each one of the nodes in L must be one of the productions in P_{out}:

$$\forall\, l \in L: \bigvee_{p \in P_{out}} (op_l = id(p)) \tag{9}$$

Input. Let I be the set of symbols that represent the inputs provided by the user. We want to guarantee that all such inputs are used in the generated programs:

$$\forall s \in I: \bigvee_{1 \le i \le n} \bigvee_{1 \le j \le k} (a_{ij} = id(s)) \tag{10}$$

Must use previous nodes. A node i must use any previous node in $PreviousHoles(i)$. Hence, one of the children must use the result of any previous node.

$$\forall 1 \le i \le n, r \in PreviousHoles(i): \bigvee_{1 \le j \le k} (a_{ij} = id(ret\_r)) \tag{11}$$

4.2 PUFS-L Framework

The PUFS-L framework integrates list manipulation operators in PUFS+. There exist 12 built-in OutSystems operators we want to synthesize, such as `ListAppend` or `ListFilter`, and a custom operator `ListMap` that is not built-in, but is included in our DSL and then compiled into OutSystems code.

The methodology chosen for the implementation was the addition of a single node of type `ExecuteAction`, which is filled by the SMT solver with the list manipulation operators.

After the implementation of the base PUFS-L framework with the chosen methodology, two additional variants were created: PUFS-L-Ordered, which aims to create a more intelligent sketch enumeration; and PUFS-L-Assisted, which builds upon the PUFS-L-Ordered framework by allowing the user to provide assistance in more complex functions.

4.2.1 *PUFS-L-Ordered Framework.*

The PUFS-L-Ordered framework has the same capabilities as the PUFS-L framework, the difference being that the sketch enumeration is guided by the input and output types.

The first change in the sketch enumeration process was filtering with the goal of minimizing the redundant attempts that could never satisfy the input/output examples. The filter consists of a set of rules, described below:

(1) If the input/output examples do not contain any element of type list, then all sketches with `ExecuteAction` nodes are skipped and the list manipulation operators are not added to the DSL.
(2) If the input/output examples have an element of type list, then at least one `ExecuteAction` node must be in the sketch.
(3) If the output is of type list, then all nodes pointing to the `End` node must be of type `ExecuteAction`.

The second change to the sketch enumerator was the sorting of sketches. From the analysis performed on real-world user examples of the OutSystem's platform in the example generation, flows with list manipulation operators usually are accompanied by other list manipulation operators and not assign and conditional ones. Thus, the sketches are sorted from the largest to the smallest amount of `ExecuteAction` nodes.

4.2.2 PUFS-L-Assisted Framework. The PUFS-L-Assisted framework introduces the possibility for the user to provide assistance in more complex functions. Operators such as `ListFilter` or `ListMap` iterate through a list and apply an operation to each element, which resulted in the need for a new type. This new type is similar to a traditional programming lambda (an anonymous function that can be dynamically defined), in that a dynamically chosen operation is performed to each element of a list. PUFS-L-Assisted allows the user to provide the lambda operations as a constant to guide the synthesizer to a more efficient search.

There are two types of lambdas: `CmpLambda` and `OpLambda`. The former allows a comparison operation to be performed to each element of a list, which is used by operators such as `ListIndexOf` and `ListAll`. The latter allows an arithmetic operation to be performed to each element of a list, which is used by the operator `ListMap`. In case the user does not provide the lambda operation as a constant, the new types `CmpLambda` and `OpLambda` can be instantiated through new operations, which are `Assign` nodes. However, this adds an extra node, which, depending on the size of the example, can greatly increase the complexity of the problem and, therefore, the time required to find a solution.

4.2.3 Changes in Implementation. Now we will present the changes in the implementation to create the different variants of PUFS-L.

Encoding variables

Remember that D is the DSL and $Prod(D)$ is the set of production rules. In the PUFS-L framework, D consists of the productions $AssignProd(D)$ and $ExecuteActionProd(D)$, i.e., $Prod(D) = AssignProd(D) \cup ExecuteActionProd(D)$. $BooleanProd(D)$, which is used to denote the set of productions that return a Boolean value, is extended to have the list manipulation operators that return a Boolean value.

Constraints

The PUFS-L framework introduces a single constraint: if a node i corresponds to an `ExecuteAction` node, then the respective symbol must be a production in $ExecuteActionProd(D)$.

$$\forall\, 1 \le i \le n : HoleType(i) = ExecuteAction \implies \bigvee_{p \in ExecuteActionProd(D)} op_i = id(p) \tag{12}$$

Sketch Enumerator

The original framework consisted in two different types of nodes: the `Assign` node and the `If` node. The new framework adds one more type of node `ExecuteAction`. The main difference is that Assign nodes may be replaced by the new node type. Thus, in the end, we create a list of sketches with all possible combinations of the nodes for each depth.

DSL and Interpreter

PUFS-L introduces a series of new types and operators, which need to be present in the DSL and have a corresponding interpreter specifying their behaviour. Thus, both the DSL and interpreter were extended to have the new values and operators.

A grammar builder was created to dynamically build the grammar from the DSL according to the type of framework configured and the input/output example types. To achieve this, we have a grammar builder with only the PUFS+ framework's values and operators, and a grammar builder with only the list manipulation's new values and operators. Then, there is a main grammar builder that, according to the configuration and example types, builds the final grammar from the individual builders. For instance, for the PUFS and PUFS+ frameworks and when the input/outputs do not have any list, the grammar must only have the values and operators of the PUFS framework, i.e., neither type list nor operators that make use of lists.

4.3 PUFS-SQL Framework

The PUFS-SQL framework combines the PUFS framework (PUFS+ version) with data aggregation capabilities. The goal of this framework is to allow the synthesis of aggregation queries using input/output examples. Two variants of PUFS-SQL were implemented: PUFS-SQL#FreeForm, which synthesizes free-form SQL queries; and PUFS-SQL#Templates, which only generates programs with queries that follow specific patterns that were observed to be highly frequent in real-world OutSystems code by the OutSystem's AI R&D team.

Similarly to PUFS-L-Ordered, ordered versions of PUFS-SQL# FreeForm and PUFS-SQL#Templates were also implemented.

The PUFS-SQL#FreeForm framework consists in the integration of an SQL synthesizer into our synthesizer. From the synthesizers presented in section 3, we decided to use the CUBES synthesizer since it seems to be the most complete in terms of the range of SQL queries supported. The new DSL has 2 new different types: `Table` and `Structure`. The former is a table that can be provided by the user. The latter is a python dictionary that corresponds to a row of a table, where the keys are the columns of the table and the values of the dictionary are the values of the row. A table with multiple rows is represented through a list of elements of type `Structure`. Besides the new types, the DSL now has new possible types that come from the CUBES specification, such as `Col` and `FilterCondition`. All of these types are generated by the CUBES framework and correspond to operators used in SQL queries.

The PUFS-SQL#Templates variant relies on an internal analysis performed on a dataset of real-world applications implemented in OutSystems. The analysis concluded that certain types of templates represent the majority of the data aggregation operations performed using the OutSystems platform. The templates that were implemented represent a total of 82.79% of all aggregates. An advantage of using templates versus the free form version is that complex operations, that would require more than one node, can be fulfilled with a single one.

Independently of the version, a new operation was added, referred to as `getStructureElement`, which retrieves the value of a column of a given `Structure` object. This operation is used in an `Assign` node. In assignment, conditional and data aggregation benchmarks, the node of type `Assign` can never be present, since the output of a query is a list of structures, which implies the need of a node of type `ExecuteAction` to obtain an element of the list before performing any assignment operations on the value. The `Assign` cannot be used before a node `DataSet`either, because the constants the SQL queries accept must be provided in the input of the specification to create the DSL values, such as the filter conditions.

4.3.1 PUFS-SQL-Ordered Framework. The PUFS-SQL-Ordered framework introduces the ordering and filtering of sketches, using the input and output types, and, similarly to the PUFS-SQL framework, it supports both the FreeForm and Templates variants.

The first change in the sketch enumeration process for the PUFS-SQL-Ordered framework was filtering, with the goal of minimizing the redundant attempts that could never satisfy the input/output examples. The filter consists of a set of rules, described below:

(1) Any sketch with nodes of type `Assign` are skipped since PUFS-SQL benchmarks do not make use of this type of node.
(2) If the input and output do not have any tables, then all sketches with `DataSet` nodes are skipped and the data aggregation operations are not added to the DSL.
(3) If the input/output examples contain a table, then at least one `DataSet` node must be in the sketch.
(4) If the output is of type list, then all nodes pointing to the `End` node must be of type `DataSet`.

The second change to the sketch enumerator was the sorting of sketches. From the analysis performed on real-world user examples of the OutSystem's platform in the example generation, flows with data aggregation operations usually were accompanied by other data aggregation operations, conditional nodes or list manipulations. Thus, considering only assignment, conditional and data aggregation nodes, the sketches are sorted from the largest amount of `DataSet` nodes to the smallest.

4.3.2 Changes in Implementation. Now we will present the changes in the implementation to create the different variants of PUFS-SQL.

Encoding variables

Remember that D is the DSL and $Prod(D)$ is the set of production rules. In the PUFS-SQL framework, D consists not only of the productions $AssignProd(D)$, but also of the productions $DataSetProd(D)$, i.e., $Prod(D) = AssignProd(D) \cup DataSetProd(D)$.

Constraints

Similarly to PUFS-L, PUFS-SQL introduces a single constraint: if a node i corresponds to a `DataSet` node, then the respective symbol must be a production in $DataSetProd(D)$.

$$\forall\, 1 \le i \le n : HoleType(i) = DataSet \implies \bigvee_{p \in DataSetProd(D)} op_i = id(p) \tag{13}$$

Sketch Enumerator

The original framework consisted in two different types of nodes: the `Assign` node and the `If` node. The new framework adds one more type of node `DataSet`. The main difference is that Assign nodes may be replaced by the new node type. Thus, in the end, we create a list of sketches with all possible combinations of the nodes for each depth.

DSL and Interpreter

Similarly to PUFS-L, PUFS-SQL introduces a series of new types and operators, which need to be present in the DSL and have a corresponding interpreter specifying their behaviour. Thus, both the DSL and interpreter were extended to have the new values and operators.

The integration with CUBES for free-form queries consisted in creating a parser that transformed our benchmarks into a format compatible with CUBES. Then, we generated the CUBES' DSL and parsed all of the values and operators obtained to our own DSL. Finally, the interpreter of CUBES was added to the list of interpreters. The decider, when verifying the input/output examples, calls the interpreter corresponding to the operator used in the solution.

Furthermore, the main grammar builder, depending on the framework configured, creates the corresponding grammar from the DSL. For instance, for the PUFS-SQL framework, the grammar should contain the operators and values of the PUFS+ framework and the SQL queries.

4.4 PUFS-X Framework

The PUFS-X framework combines all of the features of PUFS+, PUFS-L and PUFS-SQL into a single framework.

Just like for PUFS-L and PUFS-SQL, an ordered version of PUFS-X was also implemented.

4.4.1 PUFS-X-Ordered Framework. The PUFS-X-Ordered framework introduces the ordering and filtering of sketches, using the input and output types. The filtering of sketches follows the same idea as the one seen in the PUFS-L-Ordered and PUFS-SQL-Ordered frameworks, i.e., minimize the solutions that could never satisfy the input/output examples.

The filter has the following set of rules:

(1) If the input/output examples do not contain any tables, then all sketches with `DataSet` nodes are skipped and the data aggregation operations are not added to the DSL.
(2) If input/output examples do contain neither lists nor tables, then all sketches with `ExecuteAction` nodes are skipped and the list manipulation operations are not added to the DSL.
(3) If input/output examples contain a table, then at least one `DataSet` node must be in the sketch.
(4) If input/output examples contain a list and no tables, then at least one `ExecuteAction` node must be in the sketch.
(5) If the output is of type list, then all nodes pointing to the `End` node must be either of type `DataSet` or of type `ExecuteAction`.

Besides the referred set of rules, there is a verification of whether the order of nodes make sense. Nodes of type `If` are always accepted independently of where they appear. However, the remaining nodes should only be accepted if their location in the sketch makes sense. For instance, a `DataSet` node only uses input values to perform a query and never an output of another node. Thus, a `DataSet` node can always be at the beginning.

Lets start with the first node. If there are any tables in the input, then the first node should be of type `DataSet`, because it only uses as arguments the input values. If there are no tables but there are lists in the input, the first node should be either of type `ExecuteAction` or of type `Assign`, because the node of type `DataSet` will never be used when no tables are in the input. In case there are neither tables nor lists in the input, then the first node should always be of type `Assign` since there will be no need for any list operations or any SQL queries.

After the first node, if we have a node of type `DataSet` we expect to see another `DataSet` or an `ExecuteAction` node, because only these nodes can use an output of a `DataSet` node. An `Assign` node only performs operations on elements that are not lists and not tables. If we have a node of type `ExecuteAction`, we expect to see either another `ExecuteAction` node or an `Assign` node since both nodes may use each other. Finally, if we see an `Assign` node we expect another `Assign` node or an `ExecuteAction` node for the same reason.

4.4.2 Changes in Implementation. Now we will present the changes in the implementation to create the different variants of PUFS-X.

Encoding variables

Remember that D is the DSL and $Prod(D)$ is the set of production rules. In the PUFS-X framework, D consists in the productions $AssignProd(D)$, the productions $ExecuteActionProd(D)$ and the productions $DataSetProd(D)$, i.e., $Prod(D) = AssignProd(D) \cup ExecuteActionProd(D) \cup DataSetProd(D)$. $BooleanProd(D)$ denotes the set of productions that return a Boolean value.

Sketch Enumerator

The PUFS-X framework adds on the the PUFS+ framework the node types `ExecuteAction` and `DataSet`, the main difference being that the `Assign` nodes may be replaced by the new node types. Thus, we create a list of sketches with all possible combinations of the nodes for each depth.

DSL and Interpreter

With the PUFS-X framework, the DSL and interpreters do not change. However, the grammar builder adds a new configuration that creates a grammar with all operators and values mentioned thus far, i.e., PUFS+, PUFS-L and PUFS-SQL operators and values.

5 EVALUATION

Implementation

The synthesizer is implemented in Python 3.8 and it uses the Z3 SMT solver 4.8.10 with theory of Linear Integer Arithmetic. The results were obtained using an Intel(R) Core(TM) computer with an i5-8350U 1.70 GHz CPU, using a memory limit of 2 GB, running Ubuntu 20.04 LTS and with a time limit of 500 seconds.

Benchmarks

In order to evaluate our synthesizer, benchmarks are retrieved from real-world examples developed using the OutSystems platform. The benchmarks represent the different flows that our framework should be able to synthesize and is composed of 391 distinct instances. They are divided into different groups based on the type of nodes that appear in the respective solution. For example, one type of benchmark uses only assignment and conditional nodes, whereas another uses only list manipulation nodes. Then, within their group, the benchmarks are divided into different sub-groups that represent the number of nodes required by the respective solution.

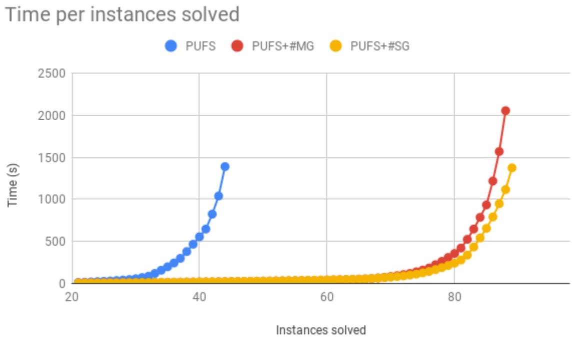

Figure 4: PUFS vs PUFS+

The goal of this experimental evaluation is to answer the following questions:

1. How does PUFS+ compare to PUFS? (section 5.1)
2. How do the Multi-Gen and Single-Gen encodings compare? (section 5.1)
3. How many, how complex and how precise can each framework solve the benchmarks?
4. How does the addition of new features affect the results of simpler benchmarks?
5. How does the pruning and ordering of sketches affect the performance of the frameworks?

5.1 PUFS+ framework

As shown in Figure 4, PUFS performs significantly worse than the PUFS+ framework, especially with sketches having 2 or more nodes. PUFS+ with the Multi-Gen encoding (PUFS+#MG) averages 23.37 seconds, and with the Single-Gen encoding (PUFS+#SG) the average lowers to 15.45 seconds per benchmark. Both encodings are able to correctly solve around 90% of the benchmarks. In contrast, PUFS averages 31.6 seconds, only being able to solve 40.45% of benchmarks. Furthermore, for the same benchmarks, PUFS spent 53.57 seconds to find the solution in contrast to the 6.1 seconds spent by PUFS+ with either encodings. These results are expected due to the changes in PUFS+ to improve the performance of the framework.

The difference between the encodings is only visible in benchmarks with more than 2 nodes, which comes as expected since 1 node and 2 node sketches have the same connectivity independently of the encoding. In 3 node benchmarks, PUFS+#MG averaged 77.28 seconds, whereas PUFS+#SG averaged 49.19 seconds. Furthermore, for the same 3 node benchmarks, PUFS+#MG spent 1617.29 seconds in contrast to the 1116.06 seconds spent by PUFS+#SG. The difference between the performance of the two types of encoding is expected, because the Single-Gen encoding forces a node to use the single previous node, whereas the Multi-Gen encoding allows solutions where any previous node can be used creating a larger search space.

PUFS had a precision of 81.81%, which means that 81.81% of solutions found were the intended ones. In contrast, both PUFS+#MG and PUFS+#SG had higher precision of 87%. Upon a closer look at the examples for each benchmark, the majority of cases where the solution found by the synthesizer was not the intended one correspond to edge cases. An example of an edge case in our synthesizer involves the operations `greater_than`

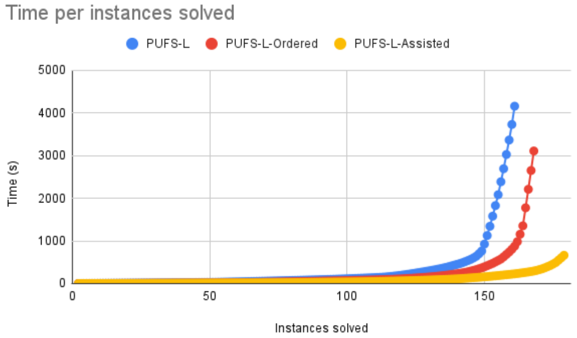

Figure 5: PUFS-L variants on list manipulation benchmarks

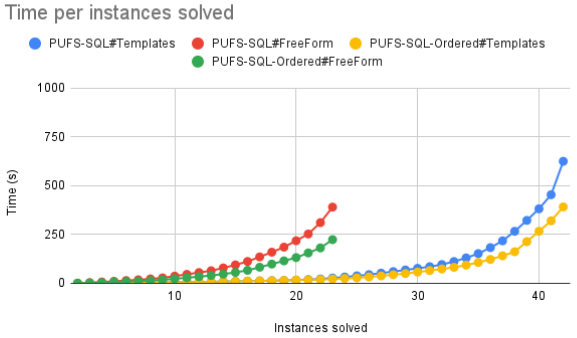

Figure 6: PUFS-SQL variants on data aggregation benchmarks

and `greater_or_equal_than` for which some examples did not specify properly the edge case that would differentiate the two operations.

5.2 PUFS-L framework

Figure 5 shows the performance of the different variants of PUFS-L running on benchmarks containing list manipulation operations, i.e., benchmarks with only list manipulation nodes and benchmarks with list manipulation, assignment and conditionals.

The impact of having the sketches pruned and ordered can be observed by comparing PUFS-L and PUFS-L-Ordered. PUFS-L averages 25.99 seconds whereas PUFS-L-Ordered averages 18.61 seconds. Besides the difference in efficiency, PUFS-L-Ordered is able to have a higher precision of 91.45% in contrast to 87.66%. Hence, the ordering and pruning has a positive impact on the framework. Similarly to the analysis of the PUFS+ framework, upon a closer look at the examples for each benchmark, the majority of cases where the solution found by the synthesizer was not the intended one corresponds to edge cases or, in the case of list manipulation, to the confusion between the operations `ListAppend` and `ListInsert`. The former is explained in section 5.1. The latter corresponds to the edge case of the operation `ListInsert`, which, when provided with an index that is higher than its size, functions as a `ListAppend` by inserting the element at the end of the list.

PUFS-L-Assisted was able to average 4.44 seconds in finding a solution with a precision of 98%. In contrast, both PUFS-L and PUFS-L-Ordered ended with a precision of 94%. PUFS-L-Assisted was also able to find all solutions. The difference in the average runtime between the PUFS-L-Ordered and PUFS-L-Assisted is not visible in every benchmark. The only benchmarks where PUFS-L-Ordered has more difficulty are the ones that require types `CmpLambda` and `OpLmabda`. This is expected since PUFS-L-Ordered does not have any assistance from the user, which means it not only needs to have an extra node to create the operation `CmpLambda` or `OpLmabda`, but it also needs to find the correct one. With the user providing the complex operation, the PUFS-L-Assisted framework is able to maintain a steady runtime throughout all benchmarks.

5.3 PUFS-SQL framework

Figure 6 shows the performance of the different variants of PUFS-SQL running on benchmarks containing data aggregation operations, i.e., benchmarks with only data aggregation nodes and benchmarks with data aggregation, assignment and conditionals. Note that there were additional benchmarks that were ran but were not portrayed in the figure that specifically targets the use cases where templates cannot find a solution. The conclusion from those benchmark results reinforced, i.e. that pruned and ordered variants have a better performance (PUFS-L-Ordered#FreeForm was more efficient than PUFS-L#FreeForm).

In Figure 6, the difference between the frameworks is evident. First, the template version performs significantly better than free-form, being able to solve every benchmark in contrast to the free-form version that only finds 56.41% of the correct solutions. The average time for PUFS-SQL#Templates was 20.49 seconds and for PUFS-SQL-Ordered#Templates was 13.09 seconds. On the other hand, PUFS-SQL#FreeForm averaged 17.74 seconds and PUFS-SQL-Ordered#FreeForm only 10.17 seconds. Despite the average being lower for the free-form version, we must note that it was only able to find 22 out of the 42 solutions and that the average time does not take into account the solutions not found within the time limit of 500 seconds. The template version is able to find all solutions and the average time takes into account all 42 benchmarks. Besides the difference between the template and free-form versions, the difference between the frameworks that are pruned and ordered compared to their respective simpler versions is clear for both the template and the free-form versions.

The total time spent on the same 21 benchmarks consolidates the conclusions, with PUFS-SQL#Templates spending 64.28 seconds, PUFS-SQL-Ordered#Templates 31.71 seconds, PUFS-SQL#FreeForm 310.92 seconds and PUFS-SQL-Ordered#FreeForm 210.56 seconds. The ordered versions are significantly more efficient and the free-form version is around 5 times worse than the template version.

We must note that for the PUFS-SQL frameworks the precision was 100%, thus showing there was no ambiguity in the benchmarks that were ran. This can be attributed to the distinct operations of the DSL for SQL queries and to the fact that the selected benchmarks did not contain edge cases.

5.4 PUFS-X framework

The PUFS-X framework was ran against every benchmark because it supports the synthesis of all features, i.e., data aggregation, list manipulation, assignment and conditionals.

Figure 7 shows the results for the best performing frameworks of PUFS, PUFS-L and PUFS-SQL, also including PUFS-X and PUFS-X-Ordered. The performance of all frameworks is similar with the exception of PUFS-X, which is clearly the worst framework. PUFS-X, without the pruning and ordering

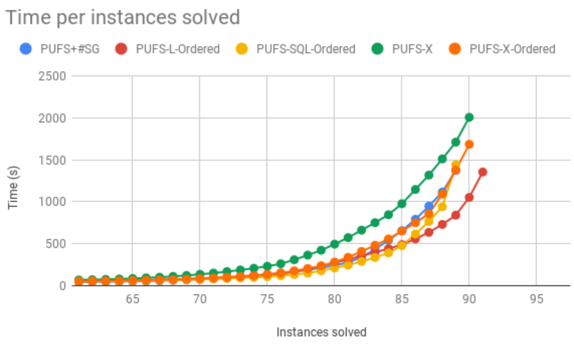

Figure 7: PUFS-SQL variants on data aggregation benchmarks

of sketches, has to enumerate through 3 sketches for depth 1, 9 sketches for depth 2 and 36 sketches for depth 3. In contrast, all of the remaining frameworks presented only have to enumerate through 1 sketch for depths 1 and 2, and 2 sketches for depth 3. With the intelligent enumeration, even though all frameworks except PUFS#SG have additional features, the impact on the performance is minimal or simply none. PUFS-L-Ordered even ended with a higher average of 14.9 seconds in contrast to the simplest framework PUFS#SG which had an average of 15.45 seconds. Knowing that both frameworks, after the pruning and the ordering of sketches are identical in their DSL and SMT constraints, the only difference is the order in which the SMT solver returns candidate solutions, which happens to show slight better results for PUFS-L-Ordered. The precision of all frameworks is similar, which is expected in an identical DSL.

In regards to list manipulation benchmarks and then data aggregation benchmarks, the results showed that PUFS-X continued to be the clear worst, showing even further how the pruning and ordering of sketches improves the performance of frameworks. For list manipulation benchmarks, PUFS-X-Ordered was able to have a similar performance to PUFS-L-Ordered since it is able to remove all sketches with `DataSet` nodes due to the lack of tables in the input. However, in data aggregation benchmarks, since `ExecuteAction` nodes can follow `DataSet` nodes, after the pruning and ordering, PUFS-X-Ordered ended with more sketches to enumerate than PUFS-SQL-Ordered, ending with a worse performance of an average of 97.21 seconds in contrast to 38.16 seconds.

All in all, for all 391 benchmarks, PUFS-X ended with an average of 37.71 seconds whereas PUFS-X-Ordered spent on average 15.4 seconds, thus reinforcing, once again, the difference in performance when the pruning and ordering of sketches is performed.

6 CONCLUSION

In this thesis, we proposed a solution to further simplify the users experience with the OutSystems platform. Our final version, PUFS-X, supports the synthesis of assignments, conditionals, list manipulations and data aggregation. The extensive evaluation performed showed us that the pruning and the ordering of sketches significantly improves the efficiency of the frameworks. Also, with the pruning, the addition of new features only affects the performance in benchmarks containing data aggregation. Hence, PUFS-X is able to solve as many benchmarks and with a similar performance as PUFS+ and PUFS-L. Furthermore, we concluded that the use of templates for SQL queries is significantly better than free-form querying.

For future work, the synthesizer can be extended to contain more features, such as loops and exception handlers. Right now, it is prepared to accept any new types of nodes with their respective DSL operators and values. The bottleneck is the exponential growth with the number of different nodes a sketch can have. Hence, it would be interesting to also see new different methods to increase the performance of the frameworks, such as a user providing a sketch that is already partially completed guiding the synthesizer to a more efficient search. Another possibility is to make use of multi-core processing and have multiple threads separately trying to find a solution.

The benchmarks used were manually created through the observation of real-world examples, creating a possible bias. It would be interesting to use real users to test the usability of the synthesizers and analyze the ambiguity generated by the examples.

REFERENCES

[1] Tidyverse. https://www.tidyverse.org/

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (nov 2018), 84–93. https://doi.org/10.1145/3208071

[3] Ricardo Brancas. 2020. *CUBES: A New Dimension in Query Synthesis From Examples*. Master's Thesis. IST - Universidade de Lisboa. https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/846778572212607

[4] C Cordell Green. 1969. 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*, Donald E Walker and Lewis M Norton (Eds.). William Kaufmann, 219–240. http://ijcai.org/Proceedings/69/Papers/023.pdf

[5] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W Hall and David A Padua (Eds.). ACM, 50–61. https://doi.org/10.1145/1993498.1993505

[6] Zohar Manna and Richard J Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (1971), 151–165. https://doi.org/10.1145/362566.362568

[7] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES: A SQL Synthesizer Using Query Reverse Engineering. *Proc. VLDB Endow.* (2020). https://doi.org/10.14778/3415478.3415492

[8] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco M Manquinho. 2019. 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 (Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.), Vol. 11802. Springer, 583–599. https://doi.org/10.1007/978-3-030-30048-7

[9] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco M Manquinho. 2019. 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 (Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.), Vol. 11802. Springer, 583–599. https://doi.org/10.1007/978-3-030-30048-7_34

[10] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science)*, Zhenjiang Hu (Ed.), Vol. 5904. Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9

[11] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. USA. https://dl.acm.org/doi/book/10.5555/1714168