TÉCNICO LISBOA

Next-Gen Pure Function Synthesis

Joana Maria Leal Coutinho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Maria Inês Camarate de Campos Lynce de Faria
Dr. Miguel Ângelo da Terra Neves

Examination Committee

Chairperson: Professor Nuno João Neves Mamede
Supervisor: Professor Maria Inês Camarate de Campos Lynce de Faria
Member of the Committee: Professor Pedro Miguel dos Santos Alves Madeira Adão

October 2021

Acknowledgments

First and foremost, I would like to thank my advisors, Professor Inês Lynce and Miguel Neves, for guiding me through this thesis from the start and always giving me the support and advice I needed. I always felt I could count on you and I am certain this chapter would have been significantly harder without you.

I would like to thank my family for all the support and patience throughout this process, always being there to listen to me and push me to be my best self. Your kind words and encouragement made the past five years possible.

I am very grateful to my friends for being there for me, especially in a time of pandemic, by always finding ways to take everyone's mind off work and other worries around us. Our time together playing games virtually made the pandemic much less lonely. I would also like to thank specifically my best friend Pedro, who I could always count on for brain storming ideas, words of encouragement and reassurance throughout this thesis.

This work was supported by OutSystems, by national funds through *Fundação para a Ciência e Tecnologia* under project UIDB/50021/2020, and project GOLEM (reference ANI 045917) funded by FEDER and FCT.

Resumo

A OutSystems é uma plataforma de automatização de software que permite aos utilizadores criar as suas próprias aplicações, utilizando interfaces gráficas em vez de uma linguagem de programação. No entanto, a lógica subjacente à interface gráfica da plataforma utiliza um grafo para ilustrar o seu comportamento, o que continua a exigir que o utilizador consiga pensar como um programador tradicional. Assim, torna-se útil para utilizadores sem essa capacidade a automatização deste mecanismo. A síntese de programas consiste na criação automática de um programa a partir de uma dada especificação. Uma função pura é uma função que retorna sempre o mesmo valor de output para o mesmo input, tendo também a propriedade de não ter efeitos colaterais.

Esta tese tem como objetivo estender o trabalho na automatização dos grafos da plataforma de OutSystems, aumentando a sua eficiência e eficácia, e permitindo operações mais complexas. A solução baseia-se na síntese de funções puras utilizando Programação-por-Exemplo como especificação e uma técnica de procura que combina Teorias do Módulo de Satisfação e enumeração de sketches.

Nesta tese, introduzimos PUFS-X, um sintetizador que suporta a geração de atribuições, condicionais, operações de listas e queries de SQL. Começamos por criar uma versão melhorada do trabalho realizado na área, o sintetizador PUFS+, que suporta a geração de atribuições e condicionais. De seguida, estendemos o sintetizador em duas formas distintas: operações de listas criando o sintetizador PUFS-L e queries de SQL, criando o sintetizador PUFS-SQL. Por fim, o sintetizador PUFS-X foi criado com todas as capacidades num só. Uma análise extensiva é realizada para observar o desempenho de cada sintetizador.

Keywords: Síntese de Programas, Teorias do Módulo de Satisfação, Programação-por-Exemplo

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. Program synthesis consists of automatically deriving a program from a specification. Pure functions always return the same value for the same input, without side effects such as altering databases.

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. 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.

In this dissertation, we introduce PUFS-X, a framework that supports generation of assignments, conditionals, list manipulation operations and data aggregation queries. We start by creating an improved version of the work done in pure function synthesis, the PUFS+ framework, which is able to synthesize programs with assignment and conditional capabilities. 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. An extensive analysis is made to observe the performance of each framework and the impact on the benchmarks when a more complex framework is used.

Keywords: Program Synthesis, Programming by Example, Satisfiability Modulo Theories

Contents

List of Figures

List of Tables

Chapter 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.

The proposed solution uses programming by example as the specification method and the search technique is a combination of sketch enumeration and satisfiability modulo theories.

1.1 Motivating 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 and the case when the value is False. The specification is presented in Table 1.3.

Id	Name	Age
1	Joao	35
2	Pedro	47
3	Matilde	31

Table 1.1: Professors

Id	Name	Age
5	Maria	25
6	Patricio	46
8	Miguel	53

Table 1.2: Support staff

According to the specification presented above, the synthesizer should be able to find a solution, such as the one seen in Figure 1.1.

1.2 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;

Input	Output
Table 1.1, Table 1.2, True	[{”Id”: 1, ”Name”: ”Joao”, ”Age”: 35}
	{”Id”: 2, ”Name”: ”Pedro”, ”Age”: 47}
	{”Id”: 3, ”Name”: ”Matilde”, ”Age”: 31}
	{”Id”: 5, ”Name”: ”Maria”, ”Age”: 25}
	{”Id”: 6, ”Name”: ”Patricio”, ”Age”: 46}
	{”Id”: 8, ”Name”: ”Miguel”, ”Age”: 53}]
Table 1.1, Table 1.2, False	[{”Id”: 1, ”Name”: ”Joao”, ”Age”: 35}
	{”Id”: 2, ”Name”: ”Pedro”, ”Age”: 47}
	{”Id”: 3, ”Name”: ”Matilde”, ”Age”: 31}]

Table 1.3: Specification

- Creation of the PUFS-L framework, which adds onto the PUFS+ framework list manipulation capabilities;
- Creation of the PUFS-SQL, which adds onto the PUFS+ framework data aggregation capabilities;
- Creation of the PUFS-X, which joins all features into a single synthesizer.

1.3 Organization

This document is organized as follows. In chapter 2 we define the relevant concepts necessary to understand the remaining of the document, such as Program Synthesis and Satisfiability Modulo Theories. Then, in chapter 3 we present existing work related to the topic of this thesis. In particular, we describe previous work on Pure Function Synthesis and the SQL synthesizers SQUARES and CUBES.

In chapter 4 we introduce our solution, starting with the PUFS+ framework, then the PUFS-L and the PUFS-SQL frameworks, and finally the PUFS-X framework that combines all features into a single synthesizer. In chapter 5 we evaluate the different configurations and versions of the frameworks using several sets of benchmarks based on real-world examples. Finally, chapter 6 concludes this document and discusses possible future work directions.

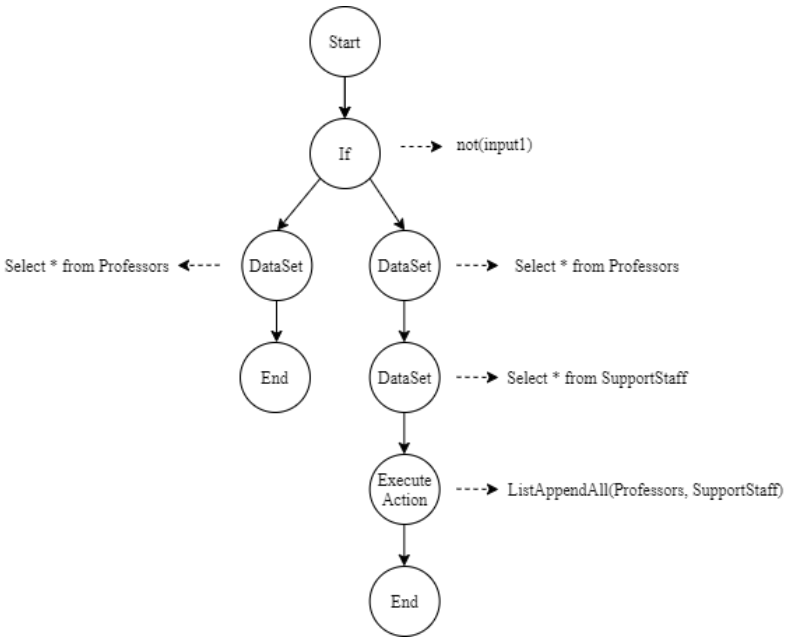


Figure 1.1: Ilustration of the motivating example

Chapter 2

Fundamental Concepts

In chapter 2, the main concepts needed to comprehend the remaining of this document are described. First, program synthesis is introduced in section 2.1, where its main challenges and techniques are specified. Then, the Programming by Examples and Satisfiability Modulo Theories are presented in sections 2.2 and 2.3, respectively. Finally, the sketching method is defined in section 2.4.

2.1 Program Synthesis

Over the years, there has been an increasing interest in Program Synthesis. We can define Program Synthesis by understanding what a program and what synthesis is: a program is a sequence of instructions that will be executed, whereas synthesis, by its definition, is the combination of components into a whole. Thus, when we refer to Program Synthesis, we are considering the task of automatically finding components of a program to reach a complete version that matches the user's intent. Consider the possibilities if we could specify our intent and have a computer figure out how to compose the corresponding program. Programming would no longer require years of study and experience, and the only real limit would be imagination.

Definition 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, as observed in Figure 2.1.

2.1.1 User specification

The first step in Program Synthesis is the specification, where the user describes the program's intended behavior.

Definition 2 (Specification)**.** Given an input $x = (x_1, x_2, ..., x_n)$ and output y, ϕ is a specification such that $\phi(x, y)$ is True, if and only if y is the desired output of x.

Figure 2.1: Program Synthesis process

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.

A formal approach is known as deductive synthesis. The main idea is to use theorem provers to build a proof from which it is possible to extract a program that satisfies the specification. Examples of this approach are the first innovative papers in the late 60s [5], and early 70s [12]. Despite these innovative early works, the required knowledge of mathematics and formulation for the user can prove to be as hard as writing the program itself.

An alternative approach is a more informal specification, known as inductive synthesis. This approach uses incomplete specifications, such as input-output examples or natural language descriptions, to specify user intent, which is considered more intuitive.

Example 1. 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 longer 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. In an informal approach, another challenge is ambiguity because examples can have multiple interpretations, depending on the context, which is difficult for a computer to comprehend correctly.

Example 2. An example of ambiguity is the interpretation of the sentence "I made her duck", which might mean "I made a duck dish for her" or "I made her crouch".

2.1.2 Program Space

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 3 (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 within and their corresponding size. Thus, if we search every possible combination, there are no guarantees of efficiency nor termination of the search. Consider the example of the Python programming language, where the

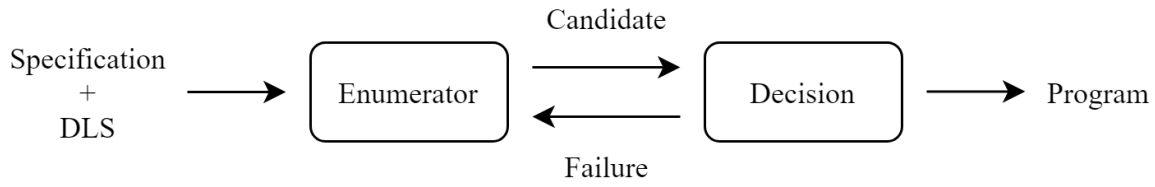


Figure 2.2: Enumerative Search process

number of possible different variables, libraries, or structures is vast. We can understand how big the program space would become.

To minimize the program space's size, instead of using full-featured programming languages such as Python, Domain Specific Languages (DSLs) are used.

Definition 4 (DSL)**.** A DSL is a language for a specialized domain, with restrictions that simplify the program space.

Example 3. 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.

$$N \rightarrow 0 \mid ... \mid 9 \mid \texttt{head}(L) \mid \texttt{last}(L) \mid \texttt{sum}(L) \mid \texttt{max}(L) \mid \texttt{min}(L)$$
$$L \rightarrow \texttt{get}(L, N) \mid \texttt{sort}(L) \mid \texttt{filter}(L, F)$$
$$F \rightarrow \texttt{geq} \mid \texttt{leq} \mid \texttt{eq}$$

A DSL is a means of balancing expressiveness and efficiency. On the one hand, we want a program space that enables the synthesis of as many programs as possible. On the other hand, if the program space proves itself too large, we lose efficiency.

2.1.3 Search Techniques

There are multiple search techniques that can be pursued, given an user specification and a program space.

Enumerative search: 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 2.2 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.

Despite the simplicity of this method, it is often a very effective strategy. However, when we scale the program space, its main obstacle is that there might not exist a good enough heuristic to efficiently reach a candidate program that satisfies the specification.

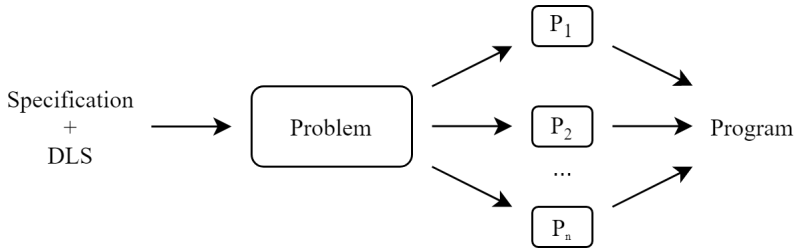


Figure 2.3: Deductive Search process

Examples of successful enumerative search algorithms are Unagi [2], an Offline Exhaustive Enumeration over the DSL program space, or the synthesizing of geometry constructions [7].

Deductive search: A deductive search is a top-down search, where the idea is to use the principle of divide-and-conquer to reduce the problem into smaller sub-problems (Figure 2.3). A certain problem P is divided into sub-problems $p_1, p_2, ..., p_n$ where each one is recursively divided into their own sub-problems such as $p_{11}, p_{12}, ...p_{1k}$. All sub-problems are then solved individually and combined by a function F into a complete working program. Being a top-down search, the deductive approach fixes the top part of an expression and then searches for its sub-expressions.

An advantage of deductive search against the enumerative approach is that, since it is a top-down search, if the grammar contains a rich set of constants, the enumerative search could get lost by guessing the constants, whereas the deductive approach can deduce which constants are correct based on the current partial program.

Examples of successful deductive search approaches are the automation of string processing in spreadsheets [6] and a framework for inductive program synthesis named FlashMeta [19].

Constraint Solving search: The constraint solving search technique involves two main steps, as seen in Figure 2.4: the generation of constraints that encode the synthesis problem and then the solving of said constraints.

The first step is the generation of a logical constraint that encodes the specification, which usually requires making assumptions regarding the unknown program's control flow. There are different possible approaches, ranging from invariant-based methods [23] to input-based [22]. On the one hand, invariant-based methods generate constraints that ensure the program is correct according to the specification. This approach has the disadvantage of often creating sophisticated constraints since the inductive invariants are often more complicated than the program itself. On the other hand, an input-based approach generates constraints that ensure a subset of inputs is satisfied, simplifying the constraints significantly but possibly compromising the user specification since informal approaches are considered incomplete. A more balanced approach is the path-based method [24], which generates constraints that assert the program satisfies the given specification on all inputs that execute a specific set of paths.

The last step is the process of solving the constraints generated, using Boolean Satisfiability (SAT) or Satisfiability Modulo Theory (SMT) solvers, which is described more deeply in section 2.3.

Examples of constraint solving techniques are, as early as 1991, the automatic inference of logical

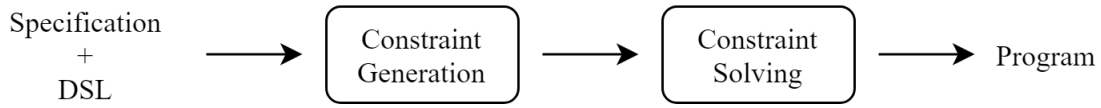


Figure 2.4: Constraint Solving process

Input	Output
[10, 29, 8, 14, 2]	29
[-1, 30, 41, 5, 89]	89
[-70, 2, 6, 3]	6

Table 2.1: Example of inputs/outputs for PBE

programs from examples [15], or, in 2013, the idea of solver-aided programming [25], which tries to reduce the complexity of the constraint generation with the framework ROSETTE. The framework receives an interpreter representing a language and develops the corresponding synthesis and verification tools.

Statistical search: The statistical search uses probabilistic models to solve the synthesis problem. These probabilistic models represent the likelihood of a function or non-terminal symbol to be used at a certain program point. Within this field, there are multiple different possible approaches, such as machine learning or genetic programming.

Machine learning is typically integrated with an enumerative or deductive search process because it allows us to calculate each possible choice's likelihood. An example of this approach is a system that tries to infer a program only with examples [13].

Genetic programming is based on biological evolution, starting with a population of individual programs and introducing random changes. Each new program is evaluated using a user-defined fitness function, and the most successful programs pass to the next phase, where the process repeats. Examples of successful uses of genetic programming are the discovery of mutual exclusion algorithms [10] or the automated repair of imperative programs [27].

2.2 Programming by Examples

One approach for specifying user intent, as seen in section 2.1.1, is inductive synthesis and, one of its sub-fields, is Programming by Examples (PBE) which relies on an input-output example based specification.

Example 4. A possible PBE specification, which represents the identification of the maximum value in the input list, can be seen in Table 2.1.

As explained in more detail in section 2.1.1, the main advantage of this approach is its simplicity for the user and, its main disadvantage, is the ambiguity that may be incurred by the usage of an incomplete specification. In this section, different approaches for dealing with these challenges are addressed.

2.2.1 Version Space Algebra

The ambiguity of input-output examples means that the feasible program space easily reaches up to several powers of ten [8], which makes it impractical to represent them explicitly. However, the possible candidates usually have several common sub-expressions, which allows us to use specialized data structures for representing several programs using a compact representation, namely Version Space Algebra (VSA).

The first definition of this concept was proposed by Tom Michell in the area of machine learning [14], which was then picked up by Tessa Lau in its application to programming by demonstration [11].

A VSA data structure is a directed graph where each node represents a set of program expressions. A leaf node is a set of expressions, whereas a parent node can be either a union of its leaves or the joint of them using an operator F. This operator can perform an intersection, clustering, ranking, or projection of leaves [19]. Using this structure, VSA allows us to encode exponential sets of candidate programs in polynomial space. Another advantage of a VSA is its ability to efficiently perform operations between leaves since these operations are proportional to the number of nodes instead of possible candidates.

2.2.2 Ambiguity Resolution

Given multiple possible program candidates for a given synthesis problem due to the ambiguity of the user's specification, the question lies in how to choose the one that is more likely to satisfy the user intent.

Ranking: One possible approach is ranking [8], where every candidate has attributed a value of likelihood, using a heuristic. The heuristic choice usually requires significant knowledge of the domain, which is retrieved from the input/output examples. There have been several attempts to automate this process, from machine learning algorithms that use the specification data [20], to machine learning algorithms that try to find features independent of the program structure, instead of relying on learned biases [4].

Active Learning: Another approach is Active Learning [8] which consists of asking the user for clarification or confirmation regarding the desired program. The traditional approach tries to find two programs that satisfy the specification, P_1 and P_2, such that for an input i, they have a different output o. Then, the user is presented with the example and chooses the correct output. Another approach includes showing the candidate programs to the user or, for non-programmers, natural language versions of them. Finally, the user can be asked to provide incorrect examples in order to prune the candidate space.

2.3 Satisfiability Modulo Theories

Given a set of Boolean variables, a propositional formula φ in Conjunctive Normal Form (CNF), is a conjuntion of clauses, where each clause is a disjunction of literals. A literal can be a variable x or its

complement $\neg x$. A *unit clause* is a clause with a single literal.

Given a propositional formula φ with n variables, the Propositional Satisfiability (SAT) problem consists in deciding whether there exists an assignment to the variables that satisfies φ.

The Satisfiability Modulo Theories (SMT) problem is a generalization of 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.

The set of *predicate* and *function* symbols, each with an non-negative arity, corresponds to a signature $\Sigma = \Sigma^F \cup \Sigma^P$, where Σ^F represents the function symbols and Σ^P represents the predicate symbols.

Predicates with 0-arity are called *propositional* symbols, and functions with 0-arity are called *constants*.

A *term* t is defined as:

$$\begin{aligned} t ::= & \, c \\ & \mid f(t_1, ..., t_n) \\ & \mid ite(\varphi, t_1, t_2) \end{aligned} \tag{2.1}$$

Where *c* and *f* are in the set of function symbols with arity 0 and arity $n > 0$ respectively and *ite* corresponds to if-then-else.

A formula φ is defined as:

$$\begin{aligned} \varphi ::= & \, A \\ & \mid p(t_1, ..., t_n) \\ & \mid t_1 = t_2 \mid \bot \mid \top \mid \neg\varphi_1 \\ & \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \\ & \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \end{aligned} \tag{2.2}$$

Where A and p are in the set of predicate symbols with arity 0 and arity $n > 0$ respectively.

Considering a theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$ of the form A, $p(t_1, ..., t_n)$, $t_1 = t_2$, $\bot$, $\top$.

On the other hand, a $\mathcal{T}$-literal is a $\mathcal{T}$-atom a or its complement $(\neg a)$ and a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals.

Given a signature $\Sigma = \Sigma^F \cup \Sigma^P$, where Σ^F represents the function symbols and Σ^P represents the predicate symbols, a Σ-*model* $\mathcal{M}$ is composed by M, a non-empty set which represents the universe of the model, and a mapping function $(\_)^M$ which maps each constant $a \in \Sigma^F$ to an element $a^{\mathcal{M}} \in M$, each function $f \in \Sigma^F$ with arity $n > 0$ to a total function $f^{\mathcal{M}} : M^n \rightarrow M$, each propositional symbol $A \in \Sigma^P$ to an element $A^{\mathcal{M}} \in \{true, false\}$, each $p \in \Sigma^P$ with arity $n > 0$ a total function $p^{\mathcal{M}} : M^n \rightarrow \{true, false\}$.

Satisfiability in SMT SMT focuses on models that belong to a given theory $\mathcal{T}$ that constrains the interpretation of the symbols in Σ.

Given a model $\mathcal{M}$, the model satisfies a formula φ if φ is true for the semantics.

A formula φ is $\mathcal{T}$-satisfiable, i.e. satisfiable in a given theory $\mathcal{T}$, iff there is an element of $\mathcal{T}$ that satisfies φ.

For given a theory $\mathcal{T}$, a formula φ is $\mathcal{T}$-satisfied by a model $\mathcal{M}$ if the model satisfies φ and $\mathcal{T}$.

So, given a $\mathcal{T}$-formula, the SMT problem consists of deciding if there is an assignment of the variables of φ such that φ is satisfied.

Example 5. Consider that $\mathcal{T}$ is the Linear Integer Arithmetic (LIA) theory.
$\phi = (x + y > 2) \wedge (x > 4) \wedge (y < 1)$, 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 $x = 5, y = 0$.

2.4 The Sketching Approach to Program Synthesis

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 a computer. Thus, Solar-Lezama introduced the concept of sketching [21, 22], 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 5 (Sketch)**.** A sketch or a partial program is a program with holes.

The core SKETCH language uses only integers to fill the holes. However, multiple syntactic extensions allow more complex sketches such as regular expression generators and repeat statements. Regular expression generators allow a much bigger set of possible variables to attribute to a hole, enabling the creation of more complex sketches. Repeat statements can be used when a programmer does not know how many iterations should be in a loop, e.g., $repeat(n)\ c$ is n repetitions of c where both n and c can have holes, allowing complex loops.

Example 6. An example of a program synthesized using SKETCH, where the holes are represented with $??$, is:

```
int function(int x):
    int result = x * ??
    assert result == x + x + x
    return result
```

A solution for this sketch is to fill the whole with the number 3 since it would validate the assertion made.

Chapter 3

Related Work

In chapter 3, previous work related to this document is discussed. First, in section 3.1, we describe the previous work done in pure function synthesis for the OutSystems platform. Then, in section 3.2, we present two synthesizers that perform SQL queries: the SQUARES and CUBES frameworks.

3.1 Pure Function Synthesis in the OutSystems Platform

Catarina Coelho proposed the first attempt at a pure function synthesizer for the OutSystems platform in her MSc thesis [1]. The proposed solution relies on input-output examples as the specification (in detail in section 2.2) and a sketch-based approach as a search technique to find the correct program efficiently (in detail in section 2.4). These techniques were integrated into the PUFS framework, as illustrated in Figure 3.1 retrieved from the thesis.

The PUFS 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.

3.1.1 Specification

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 use of PBE is expected due to the context of the problem, i.e., one of the main goals of the OutSystems platform is cut in development effort and time for the user.

3.1.2 Program space

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

[1] The MSc thesis is awaiting publication

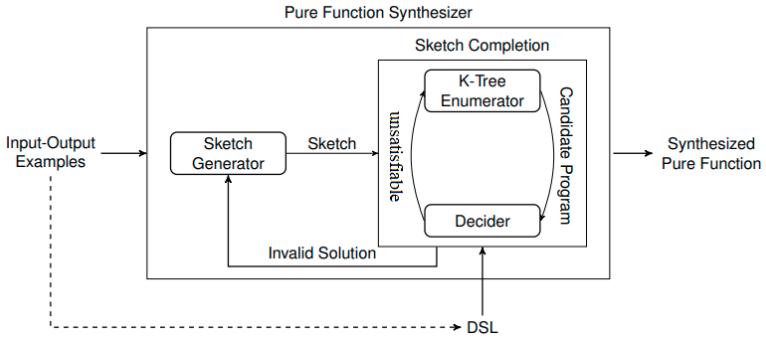


Figure 3.1: PUFS framework

variables, built-in functions or sub-expressions. The operators are unary or binary such as $+$, $-$ or $=$.

Example 7. An example of a valid OutSystems expression:

$$n + 1 \tag{3.1}$$

where both n and 1 are operands joined by the addition operator $+$.

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. The supported operators are simple Integer, Decimal, Text, and Boolean operators, such as addition, comparisons, power, length of a text, or substring matching of a text.

3.1.3 Search technique

The sketching approach is divided into two main steps: sketch generation and sketch completion. As seen in Figure 3.1, 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. An example is presented in Figure 3.2, where we have a sketch with an *Assign* node that needs to be filled.

The sketch completion step is where the holes of a sketch are filled.

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of a program, where each node is an operator and its leaves are the respective operands. An example can be observed in Figure 3.3, where there is a tree with two nodes, add and div, and three leaves, a, b and c.

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 production rules. The PUFS framework uses k-trees as the tree representation, where each k-tree represents the AST of an enumerated program, and each internal node has precisely k children. The

Figure 3.2: A partial flow

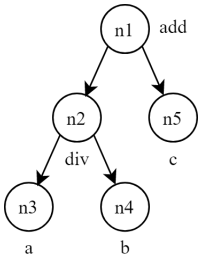


Figure 3.3: An example of an AST

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. The method chosen to encode each hole in the sketch was line-based representation [17] primarily because the sketch is represented through several trees. Another reason is because, between other methods that use the same representation, this one does not increase the number of tree nodes and, consequently, the number of variables and constraints in the SMT formula with more depth of the enumerated trees.

In Figure 3.4, we have an example of a completed sketch in continuation of the example shown in Figure 3.2, which, in this case, represents the multiplication of two input values.

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 Synthesis

One of the new features we want to implement with this thesis is data aggregation by performing SQL queries and, due to the nature of our problem, we want to use a PBE synthesizer. In this section two PBE SQL synthesizers will be presented: the SQUARES framework and the CUBES framework.

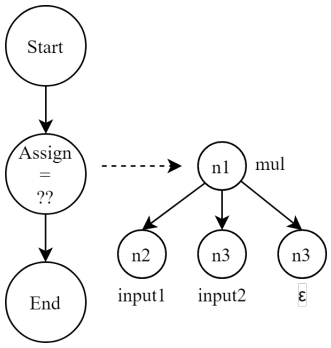


Figure 3.4: A partial flow with corresponding tree

$$
\begin{aligned}
table \rightarrow\ & input \mid \texttt{inner\_join}(table, table) \mid \texttt{inner\_join3}(table, table, table) \\
& \mid \texttt{inner\_join4}(table, table, table, table) \\
& \mid \texttt{anti\_join}(table, table) \mid \texttt{left\_join}(table, table) \\
& \mid \texttt{bind\_rows}(table, table) \mid \texttt{intersect}(table, table) \\
& \mid \texttt{filter}(table, filterCondition) \\
& \mid \texttt{filters}(table, filterCondition, filterCondition, op) \\
& \mid \texttt{summariseGrouped}(table, summariseCondition, cols) \\
tableSelect \rightarrow\ & \texttt{select}(table, selectCols, distinct) \\
op \rightarrow\ & \texttt{or} \mid \texttt{and} \\
distinct \rightarrow\ & \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Figure 3.5: DSL of the SQUARES framework

3.2.1 SQUARES

SQUARES [18] 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. According to the input/output examples and extra information, SQUARES creates the DSL variables *cols*, *filterCondition* and *summariseCondition*, which correspond to the columns used in queries, the filter conditions allowed and the summarise conditions allowed, respectively. Furthermore, the operations and remaining variables of the DSL are presented in Figure 3.5. SQUARES also supports aggregation functions, such as *sum* and *avg*.

Using the DSL described and as presented in Figure 3.6, 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. For the search, SQUARES uses the same method described for the Pure Function Synthesis in section 3.1, the line-based encoding.

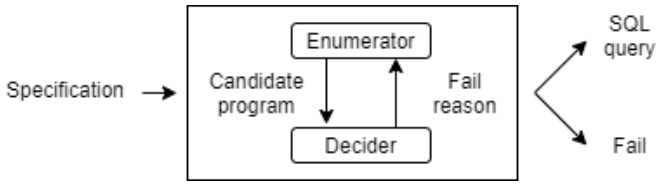


Figure 3.6: PUFS framework

$$
\begin{aligned}
table \rightarrow\ & input \mid \texttt{natural\_join}(table, table) \mid \texttt{natural\_join3}(table, table, table) \\
& \mid \texttt{natural\_join4}(table, table, table, table) \mid \texttt{left\_join}(table, table) \\
& \mid \texttt{inner\_join}(table, table, joinCondition) \\
& \mid \texttt{cross\_join}(table, table, crossJoinCondition) \\
& \mid \texttt{union}(table, table) \mid \texttt{intersect}(table, table) \\
& \mid \texttt{anti\_join}(table, table, cols) \mid \texttt{semi\_join}(table, table) \\
& \mid \texttt{filter}(table, filterCondition) \\
& \mid \texttt{summarise}(table, summariseCondition, cols) \\
& \mid \texttt{mutate}(table, summariseCondition)
\end{aligned}
$$

Figure 3.7: DSL of the CUBES framework

3.2.2 CUBES

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. This synthesizer has three different versions: sequential synthesis and two parallel synthesis variations.

CUBES-Seq is a sequential SQL synthesizer that extends the SQUARES DSL to reach a wider variety of SQL queries (the new DSL is in Figure 3.7). CUBES-Seq also supports new aggregation functions such as *n_distinct*, *str_count* and *median*. Besides the added functionality, the synthesizer introduces a new form of pruning of queries which attempts to first remove possible redundant attempts and then, from incorrect programs, learn to remove redundant future attempts. For the former, the synthesizer annotates all arguments with a pair of sets of columns in bit-vectors to enforce consistency between columns of operations. For the latter, the synthesizer looks at the number of rows of the final table to deduce which queries would not make sense to even attempt.

Example 8. For instance, the operation *filter* of a table based on a column being ”Panda” only makes sense if the value is in the table. CUBES-Seq takes this into account with the goal of removing redundant attempts. Another example of the pruning is when a program that filters a table results in a table with k rows. If we know the final program should have r rows, where $k < r$, then we can deduce that any filter condition that further reduces the number of rows by being more restrictive is always going to be an incorrect solution.

CUBES-Port is a parallel SQL synthesizer which uses portfolio solving [9] to improve the performance of the CUBES-Seq synthesizer. With the portfolio approach, the goal is to diversify the exploration of the search space by searching the same space in different ways, which can be reached by running each

thread with different SMT solver configurations.

CUBES-DC is also a parallel SQL synthesizer but instead of the portfolio approach, it tries the divide-and-conquer method, which divides the synthesis problem into several sub-problems and then solves the sub-problems in parallel. The strategy to split the program search space into sub-problems follows the work done in Propositional Satisfiability formulas [26]. Each sub-problem is represented by a cube, i.e., a sequence of operations from the DSL such that the arguments for the operations are undetermined. Then, in parallel, each cube is filled in and verified whether it is a solution. For example, a valid cube is the program *filter(inner_join(??), ??)*, which represent the problems that inner join two tables and then filter according to a condition. For this cube, both operations are filled by the possible arguments according to the DSL until the cube is either exhausted or a solution is found.

Chapter 4

Next-Gen Pure Function Synthesis

In chapter 4, the proposed solution is presented and described in detail. First, in section 4.1, we describe the PUFS+ framework, which supports generation of assignments and conditionals and builds upon the PUFS framework with the goal of improving the performance. Then, in section 4.2, the PUFS-L framework is introduced, which adds list manipulation capabilities to the PUFS framework. In section 4.3, the PUFS-SQL framework integrates the PUFS framework with an SQL synthesizer, allowing the generation of SQL queries. Then, in section 4.4 we describe the final version of the synthesizer, PUFS-X, which combines all features into a single framework. Finally, in section 4.5, the details of the implementation are described including the sketch generation process and the SMT solver constraints.

4.1 PUFS+ Framework

The initial PUFS framework had two different modes of operations: one with only assignment capabilities and one with conditional and assignment capabilities. The latter was not functioning properly because it was missing the decider step, i.e., the verification of whether a program is valid according to the specification, and the respective SMT constraints. Thus, the first step was completing the functionality and combining both modes into a single synthesizer, as intended. The synthesizer 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.

With a fully working PUFS synthesizer, several different potential improvements were identified and implemented. We refer to the improved version of PUFS as PUFS+. In the following sections, the major changes are presented. First, in section 4.1.1, we present the new types of variables the DSL contains and the expected performance impact. Then, we describe the necessity of guaranteeing the connectivity between nodes of sketches in section 4.1.2. In the section 4.1.3, three additional improvements are described: a new encoding for constants, the removal of several redundant sketches and operators and the introduction of a new configuration parameter to the framework, a rarity threshold.

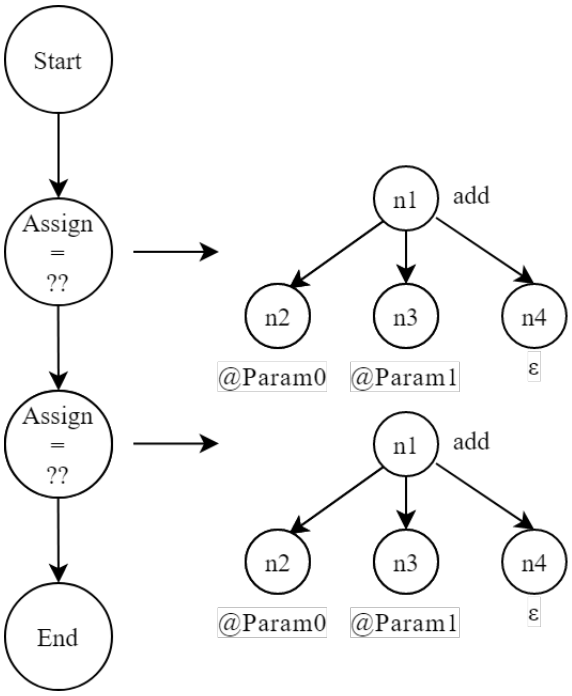


Figure 4.1: Example of the lack of connectivity between nodes in the PUFS framework

4.1.1 Fine-grained DSL Types

The PUFS framework 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.

The PUFS+ framework 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 the original PUFS framework 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 of type *Boolean*. The new versions of the operators are summarized in Tables A.1, A.2 and A.3. Note that the size of the DSL increased, since operators such as add or gt allow inputs of different types. Despite this increase, the number of combinations that the synthesizer can attempt decreases significantly, because it no longer consider operators with non-matching types.

The PUFS framework, with a specification containing only integers in its input-output examples, attempts operators such as *substr* or *not* which cannot be applied to integers. In contrast, the PUFS+ framework now defines operators according to its type. Thus, with the same specification, the only operators considered are ones that have input and outputs of the *Numeric* type. Therefore, in this scenario, the operators in Tables A.2 and A.3 are not considered since these require Text and/or Boolean values in order to be applicable.

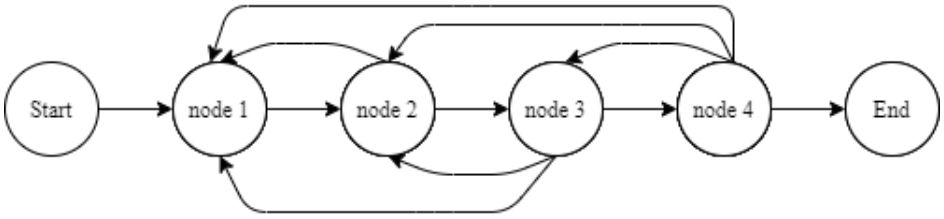


Figure 4.2: Previous nodes used in Multi-Gen encoding for the PUFS+ framework

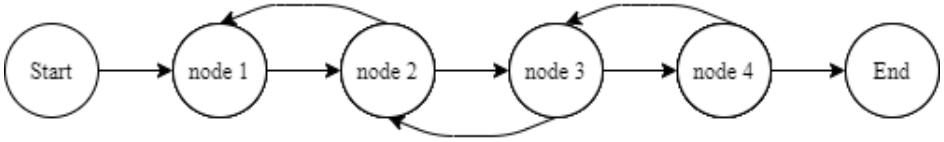


Figure 4.3: Previous nodes used in Single-Gen encoding for the PUFS+ framework

4.1.2 Node Connectivity Constraint

The PUFS framework 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 4.1, we can observe an example where the first node's operation is redundant since the following node does not use it as an argument. Note that, in order for the synthesizer to consider sketches with 2 nodes, all single node sketches must have already been exhausted. Thus, when multiple nodes are used, allowing unused nodes results in the generation of redundant programs.

The PUFS+ framework 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 4.1, 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.

Assuming the worst case scenario and not considering the improvement described in the section 4.1.1, let N be the number of attempts you can can perform for a single node, i.e., the number of combinations of the operators and inputs from the DSL, and K_d the number of combinations you can attempt that always use previous nodes, for a depth d. Both the PUFS and PUFS+ frameworks make N attempts for a 1-node sketch. For a 2-node sketch, the PUFS+ framework performs K_2 attempts. In contrast, the PUFS framework would perform the same K_2 attempts with an extra N^2 attempts where the nodes are not connected, having a total of $N^2 + K$ attempts to exhaust the search. If we consider d to be the depth of a sketch, the PUFS framework always attempts N^d more times than the PUFS+ framework in order to exhaust the search of a sketch.

The K_d combinations of the nodes can have heavily depends on whether the Multi-Gen or the Single-Gen encoding is used. With the Multi-Gen encoding, as we can observe in Figure 4.2, node 2 must use node 1, node 3 must use node 2, node 1 or both, and so forth. To get an idea of how Multi-Gen compares to Single-Gen encoding in terms of complexity, lets assume each node only uses a single previous node. In this case, with the Multi-Gen encoding and for N nodes, the last node has $N - 1$ options to use the as a previous node, then the next node has $N - 2$ options and so forth until the first node. Thus, in total, we have $(N - 1)!$ combinations to use previous nodes. In contrast, with the Single-Gen encoding as seen in Figure 4.3, each node has only one option since it must use its immediate previous node. Thus, the total number of combinations are 1. All in all, the Single-Gen encoding reduces the search space significantly, specially the more complex a sketch is, by not considering solutions that use multiple previous nodes.

4.1.3 Other Improvements

Besides the two main alterations of the framework, three additional improvements where implemented: a new encoding for constants, the removal of redundant sketches and operators and the introduction of a new configuration parameter to the framework, a rarity threshold.

Constants as Inputs

The addition of constants to the specification of the PUFS framework significantly improved its performance, since the synthesizer does not need to search for the constants itself to find the intended solution. However, when a constant is used, it requires an extra node 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.

The PUFS+ framework 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.

Pruning of Redundant Sketches and Operators

The PUFS+ framework 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 *Assign* nodes effectively return an output.

The new framework also removes redundant operators from the DSL as follows:

- The operators Lesser Than (lt) and Lesser or Equal Than (lte) can be trivially implemented using the operators Greater Than (gt) and Greater or Equal Than (gte), respectively, by simply swapping the left and right-hand sides. Thus, the Lesser Than operators were removed from the DSL, removing a total of 12 operators.

- Equal (eq) and Different ($diff$) 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, add_text_text is redundant.

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.

4.1.4 SMT constraints

This section describes how the SMT line-based encoding of the PUFS+ frameworks was adapted from the work done by Orvalho et. al. [16] 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 represents 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. To perform the enumeration of programs using a tree representation, the synthesizer encodes the tree as an SMT formula such that a model for that formula corresponds to a concrete program by assigning an operator of the DSL to each node.

Encoding Variables

Let D be the DSL. The set of production rules $Prod(D)$ in D consists of the productions $AssignProd(D)$ (presented in Tables A.1, A.2, A.3), i.e., $Prod(D) = AssignProd(D)$. Furthermore, we use $BooleanProd(D)$ to denote 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 $id : \Sigma \to \mathbb{N}_0$ be a one-to-one mapping function that maps each symbol in Σ to a unique positive identifier and $tid : Types(D) \to \mathbb{N}_0$ 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 a smaller arity 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*, which when True, the synthesizer uses the Single-Gen encoding and, when False, 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.

Constraints

.

The SMT constraints that encode the problem are as follows.

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

$$\forall\, 1 \leq i \leq n : \bigvee_{p \in Prod(D)} op_i = id(p) \tag{4.1}$$

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.

$$\forall\, 1 \leq i \leq n : HoleType(i) = If \implies \bigvee_{p \in BooleanProd(D)} op_i = id(p) \tag{4.2}$$

If a node i corresponds to an *Assign* node, then the respective symbol must be a production in $AssignProd(D)$.

$$\forall\, 1 \leq i \leq n : HoleType(i) = Assign \implies \bigvee_{p \in AssignProd(D)} op_i = id(p) \tag{4.3}$$

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

$$\forall\, 1 \leq i \leq n,\, p \in Prod(D) : (op_i = id(p)) \implies (t_i = tid(Type(p))) \tag{4.4}$$

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.

$$\forall\, 1 \leq i \leq n, r \in PreviousHoles(i),\, 1 \leq j \leq k : \bigvee_{s \in Term(D) \cup ret_r : r < i} a_{ij} = id(s) \tag{4.5}$$

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 $T(p, j) = \epsilon$.

$$\begin{gathered} \forall\, 1 \leq i \leq n,\, p \in Prod(D),\, 1 \leq j \leq arity(p),\, 1 \leq r < i : \\ ((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \implies (t_r = tid(Type(p, j))) \end{gathered} \tag{4.6}$$

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:

$$\begin{gathered} \forall\, 1 \leq i \leq n,\, p \in Prod(D),\, 1 \leq j \leq arity(p), \\ s \in \{r \in Term(D) : Type(r) \neq Type(p, j)\} : \\ (op_i = id(p)) \implies \neg(a_{ij} = id(s)) \end{gathered} \tag{4.7}$$

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:

$$\begin{gathered} \forall\, 1 \leq i \leq n,\, p \in Prod(D),\, arity(p) < j \leq k : \\ (op_i = id(p)) \implies (a_{ij} = id(\epsilon)) \end{gathered} \tag{4.8}$$

Output. Let $Type(out)$ be the type of the program's output and $P_{out} \subseteq Prod(D)$ be the subset of production rules with return type equal to $Type(out)$, i.e., $P_{out} = \{p \in Prod(D) : Type(p) = Type(out)\}$. 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}:

$$\forall\, l \in L : \bigvee_{p \in P_{out}} (op_l = id(p)) \tag{4.9}$$

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

$$\forall s \in I : \bigvee_{1 \leq i \leq n} \bigvee_{1 \leq j \leq k} (a_{ij} = id(s)) \tag{4.10}$$

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. Depending on the configuration, the $PreviousHoles(i)$ can be either the list of previous nodes from the same execution path (Multi-Gen encoding) or the last previous node (Single-Gen encoding).

$$\forall 1 \leq i \leq n, r \in PreviousHoles(i) : \bigvee_{1 \leq j \leq k} (a_{ij} = id(ret\_r)) \tag{4.11}$$

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 (described in Table A.4), such as *ListAppend* or *ListFilter*, and a custom operator *ListMap* that is not built-in, but is included in our DSL and then compiled to OutSystems code (described in Table A.5).

There are two different methodologies that could be pursued in the implementation of the new feature: add to the sketch enumeration each operator as a new node type, or add just the *ExecuteAction* node type to the sketch enumeration, which can then be filled with a concrete operator by the SMT solver. There are two different components that should be analysed in order to determine the best method: the sketch enumerator and the SMT solver.

A sketch enumerator, with N different possible nodes types and a depth d, creates N^d sketches. Currently, the PUFS framework has 2 different types of nodes. Considering L list manipulation operators, if we choose to add a new node per operator, the sketch enumerator would create $(2+L)^d$ sketches. Thus, considering the first version, for the 13 new operators we would have 15^d sketches. In contrast, with the second version we would only add one more node, ending with a total of 3^d sketches. Considering a baseline goal of depth 3, since it is the maximum depth for which PUFS+ was able to consistently provide solutions, the first version would generate $15^3 = 3375$ sketches whereas the second version would generate $3^3 = 27$ sketches.

In the first approach, the pressure on the SMT solver would be reduced since it would not need to infer the concrete operator that should be applied in the *ExecuteAction* node. However, the process of creating the constraints for the solver will need to occur regardless and already creates an impact on the performance of the synthesizer. A disadvantage for the first version compared to the second is that when the SMT solver decides how to fill in a node, it discards a series of operators due to their input and output types. The enumerator does not have that capability, which means it would unnecessarily enumerate through sketches for which there can never be a solution. To fix this, we would need to implement a way to verify whether the sketches generated matched the outputs with the next node's inputs.

Considering the impact on both the sketch enumerator and the SMT solver, the PUFS-L framework follows the methodology of the second method: add a single node of type *ExecuteAction*, which is then filled by the SMT solver with the list manipulation operators.

After the implementation of the base PUFS-L framework with the chosen methodology, two variants were created: PUFS-L-Ordered, which aims to create a more intelligent sketch enumeration (described

Depth	PUFS-L	PUFS-L-Ordered
1-node	2	1
2-nodes	4	1
3-nodes	12	2

(a) Assignment and conditional benchmarks

Depth	PUFS-L	PUFS-L-Ordered
1-node	2	1
2-nodes	4	2
3-nodes	12	5

(b) PUFS-L benchmarks

Table 4.1: PUFS-L frameworks: Number of sketches by depth

in section 4.2.1); and PUFS-L-Assisted, which builds upon the PUFS-L-Ordered framework by allowing the user to provide assistance in more complex functions (described in section 4.2.2). Independently of the version of the frameworks, it is important to note that in the case that the input-output examples do not contain elements of type *List*, the list manipulation operators are not included in the DSL. The goal is to remove all known operators from the DSL that will never be used before enumerating through the sketches.

4.2.1 PUFS-L-Ordered

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 *Execute-Action* 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 were accompanied by other list manipulation operators and not assign and conditional ones. Thus, the sketches are sorted from the largest amount of *ExecuteAction* nodes to the smallest.

Table 4.1, shows the number of sketches generated by PUFS-L and PUFS-L-Ordered for the benchmarks with and without list manipulations. The PUFS-L framework enumerates through every possible sketch, which means that, for depth 1, it generates $2^1 = 2$ sketches; for depth 2, it generates $2^2 = 4$ sketches; and for depth 3, not considering conditionals, it generates $2^3 = 8$ sketches and, considering conditionals, it generates an additional $2^2 = 4$ sketches, ending with a total of 12 sketches. In contrast, the PUFS-L-Ordered framework reduces significantly the number of enumerated sketches. For

Function Signature	Description	Examples
CreateCmpLambda(*x*: ComparisionOp, *y*: BasicType): CmpLambda	Returns a lambda with the operation *x* and value *y*.	CreateCmpLambda('$>$', 0) = lambda x: x $>$ 0 CreateCmpLambda("==", "John") = lambda x: x == "John"
CreateOpLambda1(*x*: Operation1): OpLambda	Returns a lambda with the PUFS operation *x*.	CreateOpLambda1(Abs) = lambda x: Abs(x) CreateOpLambda1(Trim) = lambda x: Trim(x)
CreateOpLambda2(*x*: Operation2Text, *y*: Text): OpLambda	Returns a lambda with the operation *x* and value *y*.	CreateOpLambda2(diff, "a") = lambda x: diff(x, "a")
CreateOpLambda2(*x*: Operation2Numeric, *y*: Numeric): OpLambda		CreateOpLambda2(mul, 2) = lambda x: mul(x, 2)
CreateOpLambda2(*x*: Operation2Boolean, *y*: Boolean): OpLambda		CreateOpLambda2(or, True) = lambda x: or(x, True)

Table 4.2: Added assign DSL operations to PUFS-L

the assignment and conditional benchmarks, the PUFS-L-Ordered framework does not consider the node *ExecuteAction* having only 1 possible sketch for the depths 1 and 2, and then 2 sketches for depth 3. For the assignment, conditional and list manipulation benchmarks, the PUFS-L-Ordered framework forces at least one *ExecuteAction* node to be present, reducing the possibilities to 1 sketch for depth 1, 2 sketches for depth 2 and a total of 5 sketches for depth 3.

The PUFS-L-Ordered framework is expected to perform more efficiently than the PUFS-L framework due to the removal of redundant sketches.

4.2.2 PUFS-L-Assisted

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 (described in Table 4.2),

Enum Signature	Description
ComparisionOp	[’>’, ’<’, ’>=’, ’<=’, ’==’, ’!=’].
Operation1	Operations of PUFS framework with one parameter.
Operation2Text	Operations of type Text of PUFS framework with two parameters.
Operation2Numeric	Operations of type Numeric of PUFS framework with two parameters.
Operation2Boolean	Operations of type Boolean of PUFS framework with two parameters..
Operation3	Operations of type Text of PUFS framework with three parameters.

Table 4.3: Added DSL enums to PUFS-L

Input	Output
[3]	[2]
[7, 3]	[6, 2]
[9, 1, 0, 3]	[8, 0, -1, 2]
[9, 0]	[8, -1]
[7, 8, 2]	[6, 7, 1]

Table 4.4: Input/output set for example 9

which are *Assign* nodes (introduced in section 4.1). 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. As we can observe, each of the lambda functions allows for a specific subset of operations (described in Table 4.3). For instance, *CreateOpLambda1* allows for any operation of the PUFS framework, which receives a single parameter.

Example 9. Lets assume the user wants a program to decrement every element of a list by one. This can be achieved using the *ListMap* operator with a lambda as defined in Table 4.2. Therefore, the user must provide a set of input/output examples, such as the one defined in Table 4.4, as well as a set of constants. Without PUFS-L-Assisted, the user must provide the constant integer 1, but, with the new framework, the user is able to provide the *OpLambda* ”sub;1”, which means an operation of subtraction with the constant 1 must be used in the final solution. The new synthesizer will no longer need an extra node to find the correct operation to perform with *ListMap* and can immediately discard a series of possible solutions that do not make use of a variable of type *OpLambda*.

.

4.2.3 Changes in Implementation

This section describes the changes in encoding variables and SMT constraints the PUFS-L framework, in the sketch enumeration process with the new type of node *ExecuteAction* and to the DSL and interpreter.

Encoding Variables

Remember that we let D be the DSL and $Prod(D)$ the set of production rules. In the PUFS-L framework, D consists of the productions $AssignProd(D)$ (presented in Tables A.1, A.2, A.3, and 4.2), and the productions $ExecuteActionProd(D)$ (presented in Tables A.4 and A.5), i.e., $Prod(D) = AssignProd(D) \cup ExecuteActionProd(D)$. Furthermore, $BooleanProd(D)$, 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.

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 $ExecuteActionProd(D)$.

$$\forall\, 1 \leq i \leq n : HoleType(i) = ExecuteAction \implies \bigvee_{p \in ExecuteActionProd(D)} op_i = id(p) \tag{4.12}$$

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.

Furthermore, 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., no type list nor any operator that makes 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. The operations introduced are described in the Table A.6 and Table A.7 and are accepted in a new type of node named *DataSet*.

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, described in section 4.3.2. Independently of the version of the frameworks, it is important to note that in the case that the input/output examples do not contain elements of type *Table*, the DSL will not have any of the data aggregation operations and only the assignment ones. The goal is to remove all known operations from the DSL that will never be used before enumerating through the sketches.

4.3.1 PUFS-SQL

The PUFS-SQL framework joins the PUFS framework (PUFS+ version) with data aggregation capabilities and is divided into two possible configurations: PUFS-SQL#freeForm and PUFS-SQL#Templates.

The PUFS-SQL#FreeForm framework consists in the integration of an SQL synthesizer into our synthesizer. From the ones presented in section 3.2, 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 integration with the DSL and interpreter is described in detail in section 4.3.3. 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*, which are described in Table A.8. 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, further described in Table A.7, 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 and, as previously established, the more we are able to minimize the number of nodes in the sketch, the more likely the framework is to find a solution. Thus, it is expected that the PUFS-SQL#Templates framework has an overall better performance than the PUFS-SQL#FreeForm framework.

Depth	PUFS-SQL	PUFS-SQL-Ordered
1-node	2	1
2-nodes	4	1
3-nodes	12	2

(a) Assignment and conditional benchmarks

Depth	PUFS-SQL	PUFS-SQL-Ordered
1-node	2	1
2-nodes	4	1
3-nodes	12	2

(b) PUFS-SQL benchmarks

Table 4.5: PUFS-SQL frameworks: Number of sketches by depth

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 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.

4.3.2 PUFS-SQL-Ordered

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.

The impact of the filtering and ordering on the number of generated sketches can be seen in Table 4.5. It shows the number of sketches generated by each framework for assignment and conditional benchmarks and then assignment, conditional and data aggregation benchmarks. With the former benchmarks, the PUFS-SQL-Ordered framework removes all sketches with data aggregation nodes

leaving only 1 sketch for 1 node and 2 node depths, and then 2 sketches for 3 nodes since it considers a sketch with a conditional node. With the latter benchmarks, the PUFS-SQL-Ordered framework also ends up having the same number of sketches, because data aggregation nodes can not be used with assignment nodes directly. Thus, the sketches generated use only data aggregation and conditional nodes. The improvement is already clear in 1 node and 2 node benchmarks, but it becomes more apparent the more complex the sketches become, such as in the 3 node benchmarks.

The PUFS-SQL-Ordered framework is expected to have a better performance than the PUFS-SQL framework due to the removal of redundant sketches.

4.3.3 Changes in Implementation

This section describes the changes in encoding variables and SMT constraints the PUFS-SQL framework, in the sketch enumeration process with the new type of node *DataSet* and to the DSL and interpreter.

Encoding Variables

Remember that we let D be the DSL and $Prod(D)$ the set of production rules. In the PUFS-SQL framework, D consists not only of the productions $AssignProd(D)$ (presented in Tables A.1, A.2, A.3), but also of the productions $DataSetProd(D)$ (presented in Tables A.6 and A.7), 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 an *DataSet* node, then the respective symbol must be a production in $DataSetProd(D)$.

$$\forall\, 1 \leq i \leq n : HoleType(i) = DataSet \implies \bigvee_{p \in DataSetProd(D)} op_i = id(p) \tag{4.13}$$

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, which was introduced in the PUFS-L framework (section 4.2.3), 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.

We must note that the constants used in SQL queries must be provided in the input of the specification because they are used to create the DSL values, such as the filter conditions. Thus, there cannot be a node of type *Assign* that performs an operation on an input that is then used in a node of type *DataSet*. Therefore, all benchmarks with only *Assign*, *If* and *DataSet* nodes will never have a *Assign* node connected to an *DataSet* node.

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 and is further described in section 4.4.1.

4.4.1 PUFS-X-Ordered

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 not contain any list nor any 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 of type *DataSet* or of type *ExecuteAction*.

Depth	PUFS-X	PUFS-X-Ordered
1-node	3	1
2-nodes	9	1
3-nodes	36	2

(a) Assignment and conditional benchmarks

Depth	PUFS-X	PUFS-X-Ordered
1-node	3	1
2-nodes	9	2
3-nodes	36	6

(b) PUFS-X benchmarks

Table 4.6: PUFS-X frameworks: Number of sketches by depth

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, as explained in section 4.3.3, a *DataSet* node only uses input values to perform a query and never an output of another node. Thus, a *DataSet* node must 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 no 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 nor 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.

With the conjunction of the initial rules and the path verification, we remove a significant amount of sketches that would never result in a valid solution. In Table 4.6 we can observe the difference in the number of sketches for assignment and conditional benchmarks and then for PUFS-X bechmarks, i.e., benchmarks that use assignment, conditional, list manipulation and data aggregation nodes. Without considering nodes of type *If*, the number of sketches for a depth d is N^d sketches. Thus, as we can see, the PUFS-X framework generates 3 sketches for depth 1 and 9 sketches for depth 2. With depth 3, besides the $3^3 = 27$ sketches, we have an additional 9 sketches that use *If* nodes. With the filtering of the PUFS-X-Ordered framework, for the assignment and conditional benchmarks, the number of sketches reduces drastically specially for depth 3, where instead of 36 sketches the synthesizer only needs to enumerate 2. For the PUFS-X benchmarks, the difference is still very significant, with depth 3 resulting in 6 sketches instead of the 36.

4.4.2 Changes in Implementation

This section describes the changes in encoding variables the PUFS-X framework, in the sketch enumeration process and to the DSL and interpreter.

Encoding Variables

Remember that we let D be the DSL and $Prod(D)$ the set of production rules. In the PUFS-X framework, D consists in the productions $AssignProd(D)$ (presented in Tables A.1, A.2, A.3 and 4.2), the productions $ExecuteActionProd(D)$ (presented in Tables A.4 and A.5) and the productions $DataSetProd(D)$ (presented in Tables A.6 and A.7), i.e., $Prod(D) = AssignProd(D) \cup ExecuteActionProd(D) \cup DataSetProd(D)$. Furthermore, $BooleanProd(D)$, which is used to denote the set of productions that return a Boolean value, is has all operators that return a Boolean.

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.

4.5 User input

In this section, the user input in regards to the configuration of the synthesizer and the benchmarks is presented.

Synthesizer's configuration

The synthesizer is configured through a JSON file provided by the user, which contains possible customizations and the directory containing the benchmarks.

The first set of configurations for the user is the *sketch_config*, i.e., the different hyper-parameters that can be customized in the sketch enumeration process. The first hyper-parameter is the *sketch_type*, which indicates the types of nodes that are allowed in the sketches, such as only *Assign* nodes or all the combinations of nodes. The *sketch_min* and the *sketch_max* indicate the minimum/maximum depth of the sketches, respectively. Finally, the *pruning* hyper-parameter can be either true or false, indicating whether the sketches should be ordered and filtered.

```
{
    "sketch_config": {
        "sketch_type": "sketch_x",
        "sketch_min": 2,
        "sketch_max": 6,
        "pruning": true
    },
    "version": "PUFS-X",
    "single_gen": false,
    "free_form_sql": false,
    "DSL_USE_TH": 0,
    "DEBUG": false
}
```

Figure 4.4: Example of a configuration

The remaining configurations are the *version*, which corresponds to the version of the framework to be executed (e.g. PUFS+, PUFS-X) and *free_form_sql*, which is true if one wishes to generate free-form SQL instead of the templates described in section 4.3.

Example 10. An example of a configuration is presented in Figure 5.4, where the PUFS-X framework is set to run on the PUFS benchmarks with the following settings: sketches contain all types of nodes; sketches have a minimum depth of 2 and maximum depth of 6; sketches have a maximum of 4 nodes; and pruning is activated. Furthermore, neither the free-form nor the debug are activated.

Benchmarks specification

A benchmark is provided in a *yaml* file, where the input and output examples are described and constants are provided to assist the synthesizer. For the synthesizer to provide a solution it requires a set of input and output examples, which can range from one example to many more. The number of examples needed for a program depends on the complexity of the desired solution, because the examples need to be sufficient to capture all kinds of corner cases of the desired solution. For instance, if a program should perform an operation depending on whether a number is bigger than another, then at least 3 examples should be given: one for the lower limit, one for the higher limit and one for the point of the change, which differentiates whether the condition includes or excludes the point of comparison.

Chapter 5

Evaluation

In chapter 5, the frameworks are evaluated and the results analysed. First, in section 5.1, the benchmark sets used in the evaluation and respective source are described. Then, in section 5.2, the method of evaluation used is detailed. Finally, in section 5.3, an analysis is performed on the experimental results obtained for each framework and their respective versions.

5.1 Benchmark Description

In order to evaluate our synthesizer, a set of benchmarks were retrieved from real-world examples developed using the OutSystems platform. The set of benchmarks represents the different flows that our framework should be able to synthesize, and are divided into different groups based on the type of nodes that appear in the respective solution. For example, one set of benchmarks requires only assignment and conditional nodes, whereas another set of benchmarks requires 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. The latter separation allows us to observe how the synthesizer behaves as the complexity of the problem increases.

The first set of benchmarks requires only assignment and conditional nodes to be synthesized and will be further described in the subsection 5.1.1. These benchmarks were created using real examples of the OutSystems platform. However, the remaining benchmarks are not specific examples, but instances of operations based on the examples observed in the platform. In the following sub-sections the different types of benchmarks are described. Table 5.1 shows an overview of all the benchmarks used.

5.1.1 Assign and conditional nodes

The examples gathered with assign and conditional nodes were selected from the examples collected by Catarina Coelho for evaluating the PUFS framework. These benchmarks only contain nodes of type *Start*, *End*, *Assign* and *If* and the operations used within the nodes are limited to the ones accepted by the PUFS framework (described in Tables A.1, A.2 and A.3).

Benchmarks	# Instances
Assign and conditional	97
List manipulation	127
Assign, conditional and list manipulation	52
Data aggregation	46
Assign, conditional and data aggregation	10
List manipulation and data aggregation	35
Assign, conditional, list manipulation and data aggregation	24
Total	391

Table 5.1: Summary of benchmarks used

A total of 97 examples were selected as benchmarks, ranging from examples containing 1 node up to 3 nodes. We observed that examples with more than 3 nodes were too difficult for the synthesizer to solve within the pre-determined runtime limit. The evaluation presented throughout this chapter will demonstrate the results for examples of 1 node, 2 nodes, 3 nodes and then all together in order to better understand the evolution of the performance of the synthesizer.

This set of benchmarks is used to evaluate several different frameworks, from the original PUFS framework to more complex ones that use list manipulation and aggregation capabilities. The goal is to evaluate the impact of using more complex frameworks on the efficiency and effectiveness.

Example 11. Examples of possible assign and conditional benchmarks are: add(input1, input2), which requires 1 node of type *Assign*; replace(toLower(input1), input2, input3), which requires 2 nodes of type *Assign*; and If(gt(input1, input2), sub(input1, input2), sub(input2, input1)) (observed in Figure 5.1), which requires 2 nodes of type *Assign* and 1 node of type *If*.

5.1.2 List manipulation nodes

The examples gathered with list manipulation nodes were created manually based on real-world OutSystems examples to represent the different operations the PUFS-L framework is capable of solving. Despite the examples being created manually, it was after an extensive search through OutSystems real-world examples. These benchmarks contain nodes of type *Start*, *End* and *ExecuteAction* and the operations used are described in Tables A.4 and A.5.

The resulting benchmark set contains 127 benchmarks that are divided, similarly to the assign and conditional benchmarks, into three groups from 1 node to 3 node solutions. This set of benchmarks is used to evaluate the different versions of the PUFS-L framework and the final version of the synthesizer, the PUFS-X framework.

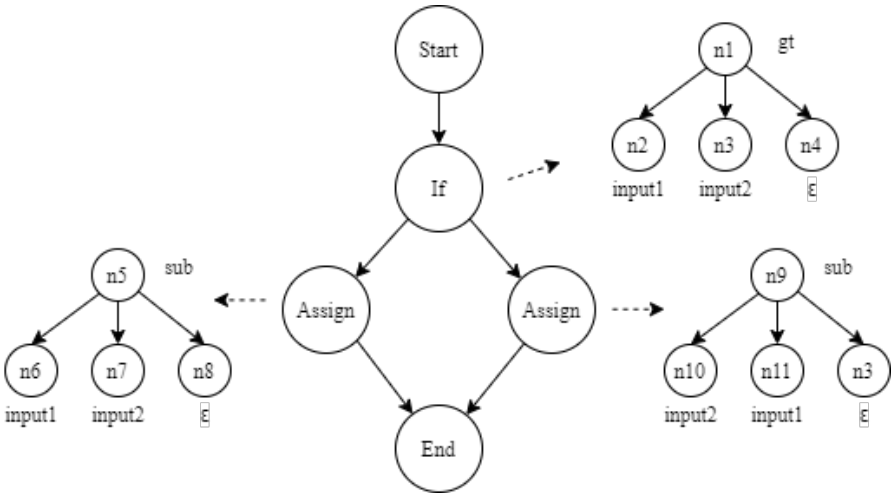


Figure 5.1: Example of a PUFS benchmark

Example 12. Examples of possible list manipulation benchmarks are: ListAppend(input1, input2), which requires 1 node of type *ExecuteAction*; ListAppendAll(ListAppendAll(input1, input2), input3), which requires 2 nodes of type *ExecuteAction*; and ListAny(ListFilter(ListAppend(input1, input2), input3), input4), which requires 3 nodes of type *ExecuteAction*.

5.1.3 Assign, conditional and list manipulation nodes

The examples gathered with assign, conditional and list manipulation nodes were created manually based on real world OutSystems examples to represent the different operations the PUFS-L framework is capable of solving. Similarly to the list manipulation nodes, despite the examples being created manually, it was after an extensive search through OutSystems real-world examples. These benchmarks contain nodes of type *Start*, *End*, *Assign*, *If* and *ExecuteAction* and the operations used are described in Tables A.1 to 4.3.

The resulting benchmark set contains 52 benchmarks separated into two groups: 2 node and 3 node solutions. Since this is the combination of assign, conditional with list manipulation nodes, benchmarks with 1-node are already represented in the sets presented in sections 5.1.1 and 5.1.2. These benchmarks are used to evaluate the different versions of the PUFS-L framework and the final version of the synthesizer, the PUFS-X framework.

Example 13. Examples of possible assign, conditional and list manipulation benchmarks are: ListAppend(input1, add(input2, input3) (observed in Figure 5.2), which requires 1 node of type *Assign* and 1 node of type *ExecuteAction*; ListAppend(input1, replace(toLower(input2), input3, input4)), which requires 2 nodes of type *Assign* and 1 node of type *ExecuteAction*; and If(ListAny(input1, input2), ListAppend(input1, mul(input3, input4)), ListAppend(input1, div(input3, input4))) , which requires 2 nodes of type *Assign*, 1 node of type *ExecuteAction* and 1 node of type *If*.

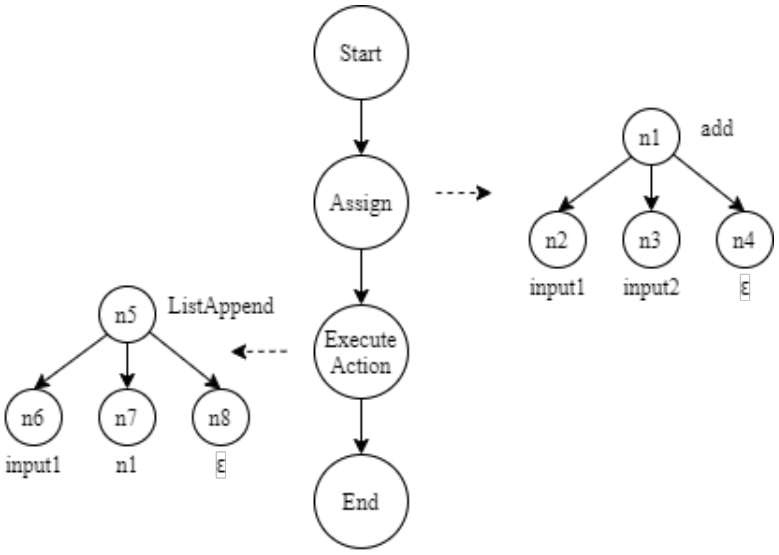


Figure 5.2: Example of a PUFS-L benchmark

5.1.4 Data aggregation nodes

The examples gathered with data aggregation nodes were created manually based on real world OutSystems examples to represent the different operations the PUFS-SQL framework is capable of solving. Despite the examples being created manually, it was after an extensive search through OutSystems real-world examples. These benchmarks contain nodes of type *Start*, *End* and *DataSet* and the operations used are described in Tables A.6 and A.7.

The resulting benchmark set contains 46 benchmarks that are divided into two groups: template benchmarks and free-form benchmarks. The former corresponds to benchmarks that have solutions with the templates defined in the PUFS-SQL-Templates (described in section 4.3.1). The latter corresponds to benchmarks whose solution does not have a template. The sets of benchmarks are evaluated against the different versions of the PUFS-SQL frameworks and the final version of the synthesizer, the PUFS-X framework.

Example 14. Examples of possible data aggregation template benchmarks are: Select * from table1; Select * from table1 where name == ”John”; and Select * from join(table1, table2) where name == ”John”. The number of nodes required for each example will depend on the configuration used (with templates or free-form). Examples of possible data aggregation free-form benchmarks are: Select avg(age) from table1; Select sum(age) from table1; and Select * from join(table1, table2).

5.1.5 Assign, conditional and data aggregation nodes

The examples gathered with assign, conditional and data aggregation nodes were created manually based on real world OutSystems examples to represent the different operations the PUFS-SQL framework is capable of solving. These benchmarks contain nodes of type *Start*, *End*, *If* and *DataSet* and the operations used are described in Tables A.1, A.2, A.3, A.6 and A.7. Benchmarks with the node *Assign* will not be added since assign operations can not occur directly on the output of a query, i.e., a query

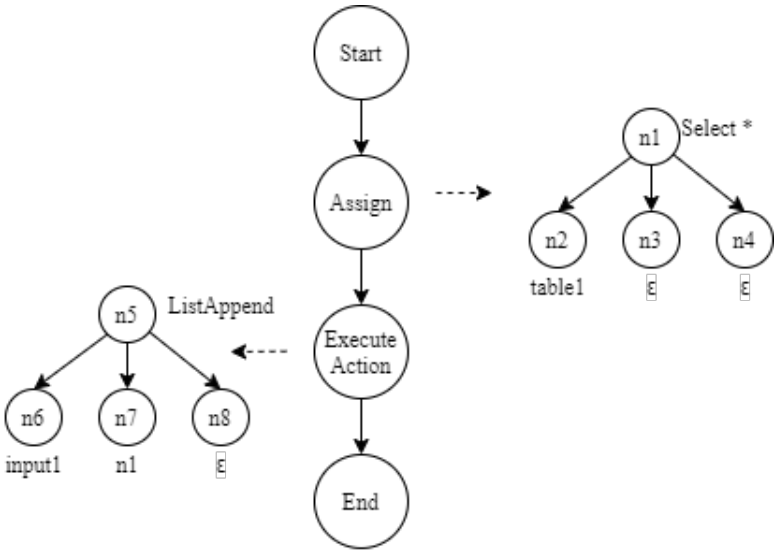


Figure 5.3: Example of a list manipulation and data aggregation benchmark

outputs a list of structures and if we want to perform an operation over an element of the list we would always need an *ExecuteAction* node.

The resulting benchmark set contains 10 benchmarks whose solution is an if and then a query depending on a condition. The set of benchmarks is evaluated against the different versions of the PUFS-SQL frameworks and the final version of the synthesizer, the PUFS-X framework.

Example 15. Examples of possible assign, conditional and data aggregation benchmarks are: If(condition, Select * from table1, Select * from table2); and If(condition; Select * from join(table1, table2) where name == ”John”, Select * from join(table1, table2) where name == ”Peter”). Similarly to the data aggregation benchmarks, the number of nodes required for each example will depend on the configuration used (with templates or free-form).

5.1.6 List manipulation and data aggregation nodes

The examples gathered with list manipulation and data aggregation nodes were created manually based on real world OutSystems examples to represent the different operations the PUFS-X framework is capable of solving. These benchmarks contain nodes of type *Start*, *End*, *ExecuteAction* and *DataSet* and the operations used are described in Tables from A.4 to A.7. Benchmarks with the node *Assign* will not be added since assign operations can not occur directly on the output of a query.

The resulting benchmark set contains 35 benchmarks that are divided into two groups: 2-node and 3-node complexity. The sets of benchmarks are evaluated against the different versions of the PUFS-X frameworks.

Example 16. Examples of possible list manipulation and data aggregation benchmarks are: ListAppend(Select * from table1, input1) (seen in Figure 5.3); and ListDistinct(Select * from table1, input1). Similarly to the data aggregation benchmarks, the number of nodes required for each example will depend on the configuration used (with templates or free-form). However, with the template configuration

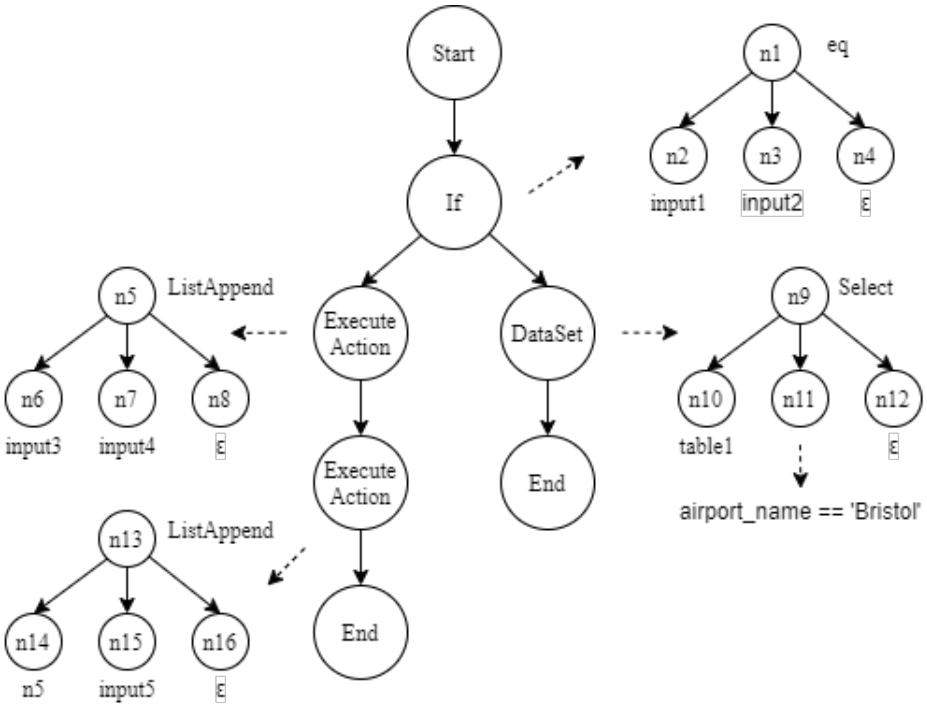


Figure 5.4: Example of a PUFS-X benchmark

for example, the first example needs 1 node of type *DataSet* and one node of type *ExecuteAction* as seen in Figure.

5.1.7 Assign, conditional, list manipulation and data aggregation nodes

The examples gathered with assign, conditional and data aggregation nodes were created manually based on real world OutSystems examples to represent the different operations the PUFS-SQL framework is capable of solving. These benchmarks contain nodes of type *Start*, *End*, *Assign*, *If*, *ExecuteAction* and *DataSet* and the operations used are described in Tables from A.1 to A.7.

The resulting benchmark set contains 24 benchmarks that are divided into two groups: 3-node and 4-node complexity. The sets of benchmarks are evaluated against the different versions of the PUFS-X framework.

Example 17. Examples of possible assign, conditional, list manipulation and data aggregation benchmarks are: eq_numeric_numeric(getElementList(ListMap(Select * from table1, input1), input2), input3), which needs a node *DataSet*, then two nodes *ExecuteAction* and, finally, one node *Assign*; and If(eq(input1, input2), Select * from table1 where airport_name == 'Bristol', ListAppend(ListAppend(input3, input4), input5)) (seen in Figure 5.4), where we need a node *If*, two nodes *ExecuteAction* and a node *DataSet*.

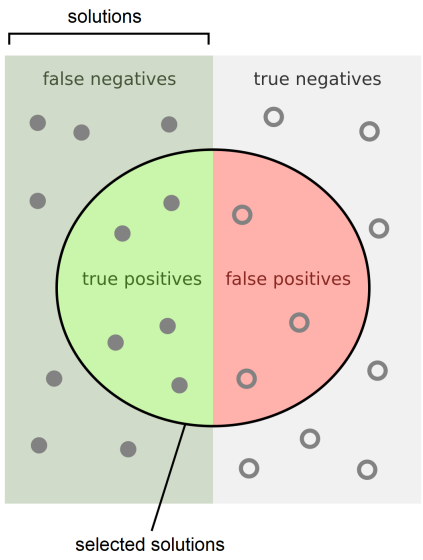


Figure 5.5: Precision and recall metrics[1]

5.2 Evaluation Method

The first evaluation metric used is efficiency, which represents the average runtime it takes for the synthesizer to produce a solution. This metric allows us to evaluate not only the impact of the improvements of the frameworks, but also the impact of adding new features.

Then, we have the metrics precision and recall, which take in consideration true positives, false positives, true negatives and false negatives (represented in Figure 5.5). For our evaluation, a true positive is when the synthesizer provides the intended solution; a false positive is when the synthesizer provides a solution, but due to the ambiguity of the specification, not the intended one; a true negative is when there is no solution and the synthesizer does not provide one (this will never occur in our evaluation, because we will only run benchmarks with frameworks that have a solution); finally, a false negative is when the synthesizer should have found a solution, but it wasn't able to within the pre-determined runtime limit.

The precision metric is calculated through the equation 5.1 and represents the percentage of solutions that match the specification of the user. The value ranges from 0 to 1. The closer the value is to 1, the more solutions are the ones intended by the user. On the other hand, the closer the value is to 0, the more the solutions are valid according to the specification, but not the ones intended by the user.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{5.1}$$

The recall metric is calculated through the equation 5.2 and refers to the percentage of total solutions correctly classified by the synthesizer. Similarly to the precision metric, the value ranges from 0 to 1. The closer the value is to 1, the more intended solutions where found relative to the problems that could not be solved in the pre-determined runtime limit. On the other hand, the closer the value is to 0, the

[1] the image was adapted from https://en.wikipedia.org/wiki/Precision_and_recall

more are the problems that were not able to be solved in relation to the ones that were found.

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{5.2}$$

Finally, a last metric will be used, which distinguishes from the previous three evaluation methods since it does not characterize all benchmarks, but only the ones where the frameworks returned the same solution. This metric corresponds to the total runtime spent by each framework on the same benchmarks.

5.3 Experimental Results

In this section, we discuss the different frameworks created throughout the thesis. First, we have an analysis of the PUFS framework, where we compare the original framework PUFS with the improved version PUFS+. Then, we have the PUFS-L, PUFS-SQL and PUFS-X frameworks, where we present the main different versions of each framework and observe the differences in the performance.

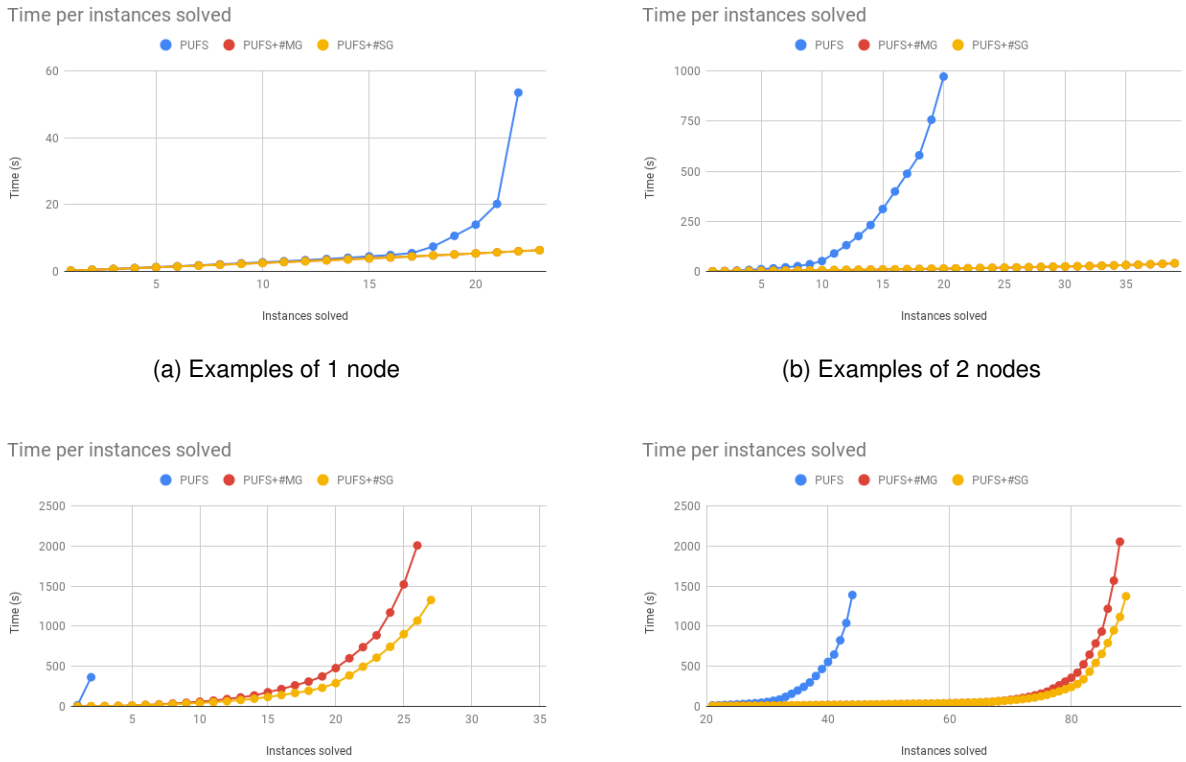
For each framework and complexity of the benchmark, we present the results in a figure with graphics, which correspond to the number of instances solved (x axis) by the runtime passed (y axis). Then, we have a table where the average runtime and the metrics precision and recall are presented. Finally, we have a table with the total runtime spent on the same benchmarks for each framework. The goal of this experimental evaluation is to answer the following questions:

1. How does PUFS+ compare to PUFS? (section 5.3.1)
2. How does the Multi-Gen and Single-Gen encoding compare? (section 5.3.1)
3. How many, how complex and with which accuracy 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?

Implementation The synthesizer is implemented in Python 3.8 and it uses the Z3 SMT solver 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.

5.3.1 PUFS+ framework

The PUFS, the PUFS+#MG and the PUFS+#SG framework were run with the gathered benchmarks of only assign and conditional nodes, where the respective results are shown in Figure 5.6, Table 5.2 and Table 5.3. First we discuss the benchmarks according to the number of nodes required by the respective solutions, and then we show an overall perspective of the results.



(a) Examples of 1 node

(b) Examples of 2 nodes

(c) Examples of 3 nodes

(d) Examples of 1, 2 and 3 nodes

Figure 5.6: PUFS framework versions: Runtime per instances solved

1-node benchmarks

As observed in Figure 5.6a, the PUFS framework immediately shows some difficulty in finding solutions for the 1-node benchmarks, whereas the PUFS+#MG and PUFS+#SG frameworks maintain a steady runtime throughout the benchmarks. The performance of the PUFS+#MG and PUFS+#SG frameworks are expected to have the similar results for 1 node benchmarks and only start to differ with multiple nodes, because the difference in encoding only applies starting from 3 nodes.

In Table 5.2, we have the average runtime it takes for the frameworks to find a solution and we observe that the PUFS framework averages around 2.435 seconds whereas the PUFS+#MG and the PUFS+#SG frameworks maintain an average around 0.28 seconds. The difference in efficiency occurs in 5 out of the 24 benchmarks which require significantly more runtime for the PUFS framework than the remaining frameworks. Upon a closer look at the examples, we concluded that this is due to the usage of constants, which explains the spike in the runtime to find a solution, because, as described in section 4.1.3, the PUFS framework needs an extra node per constant used, whereas the PUFS+ frameworks do not have this limitation.

The precision metric being 1 for all frameworks shows us that, when the frameworks were able to find a solution, it was the intended one. This is expected since we are working with 1-node benchmarks and all the functions of the DSL perform different operations. Ambiguity is only expected to be observed with more complex benchmarks.

The PUFS framework achieves a recall of 0.9565, because it was unable to find a solution for 1 of

	PUFS			PUFS+#MG			PUFS+#SG		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	2.435	1	0.9565	0.2770	1	1	0.2781	1	1
2-nodes	48.6455	0.7	0.4242	1.0410	0.8205	1	1.0444	0.8205	1
3-nodes	182	0	0	77.2773	0.8846	0.8519	49.1863	0.8519	0.7419
all nodes	31.6018	0.8181	0.4045	23.3657	0.875	0.8953	15.4507	0.8652	0.9059

Table 5.2: Evaluation metrics for the PUFS frameworks

	# Benchmarks	PUFS (s)	PUFS+#MG (s)	PUFS+#SG (s)
1-node	22	53.57	6.1	6.1
2-nodes	14	839.48	16.58	16.64
3-nodes	19	–	1617.29	1116.06

Table 5.3: Total runtime of same benchmarks for the PUFS frameworks

the 24 benchmarks. As expected, this benchmark uses constants in the specification. Moreover, it uses 2 constants, which implies the need of 2 extra nodes to find a solution. The PUFS+ frameworks were able to reach a recall of 1, finding the intended solutions to all benchmarks.

All in all, in 1-node benchmarks, performance differences are only noticeable when constants are used. However, as is shown in Table 5.3, for the 22 benchmarks all of the frameworks were able to reach the correct solution, the PUFS framework showed a much worse performance needing a total of 53.57 seconds, in contrast to the 6.1 seconds it took the PUFS+ frameworks.

2-node benchmarks

With 2-node benchmarks, as can be observe in Figure 5.6b, the PUFS framework struggles to find solutions for several instances. In contrast, the PUFS+ framework continues to maintain a steady runtime for all benchmarks independently on the encoding.

In Table 5.2, we can observe that the average runtime to find a solution for the PUFS framework is 48.6455 seconds, whereas for the PUFS+ frameworks we have an average around 1.04 seconds. In Table 5.3 the difference between the performance of the PUFS framework against the PUFS+ frameworks is even more noticeable, with the PUFS framework taking a total of 839.48 seconds to solve 14 benchmarks and the PUFS+ frameworks only taking around 16.6 seconds to solve the same benchmarks. The significant difference can no longer be just attributed to the use of constants, but also to the pruning of invalid programs performed by PUFS+ that is enabled by the node connectivity constraints and the improved DSL. The difference between the Multi-Gen and Single-Gen encoding is not visible yet, which is expected since, independently of the encoding, both frameworks must have the second

node use the first node. A difference is expected starting with 3 node sketches.

The precision metric for the PUFS framework is at 0.7, which shows that, between the solutions found, 30% were not the intended ones. This is due to the ambiguity of the specification. The PUFS+ framework achieves a higher precision of 0.8205 for both encodings.

The recall metric maintains at 1 for the PUFS+ frameworks showing us that it was able to find a solution for every benchmarks. However, the PUFS framework was only able get a recall of 0.4242, since it was not able to find a solution for 19 out of the 39 benchmarks.

3-node benchmarks

For the 3-node benchmarks, the PUFS+ framework starts to show some difficulty in finding solutions, as we can see in Figure 5.6c. We also start to see the difference in performance between the two encodings of the PUFS+ framework (Multi-Gen and Single-Gen encoding).

As expected, the PUFS framework performs significantly worse in every metric, as we can observe in Table 5.2. Out of the two benchmarks it was able to solve, it took an average of 182 seconds. With the complexity of these benchmarks, the PUFS+ frameworks made a drastic jump of runtime performance from 1.04 seconds to 77.2773 seconds in the case of the Multi-Gen encoding and to 49.1863 in the case of Single-Gen encoding. This drastic jump is not surprising due to the combinatorial nature of the problem, i.e., the exponential complexity of synthesis. 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 to be used creating a larger search space. The difference can be observed further in Table 5.3, where, for the same benchmarks, the PUFS+#MG framework spent a total of 1617.29 seconds in contrast to the 1116.06 seconds it took the PUFS+#SG framework.

The solutions obtained from the PUFS framework were not the intended ones leaving the precision and recall at 0 for this framework. In contrast, the PUFS+#MG framework had a precision of 88.46% and a recall of 85.19%, which shows us that most solutions found were the intended ones and that 9 benchmarks were not able to reach a solution. The PUFS+#SG framework had a slight worse precision of 85.19% and a worse recall of 74.19%. However, it was able to find one more solution than the PUFS+#MG framework.

All benchmarks

In Figure 5.6d and in the last row of Table 5.2, the results for all the benchmarks are shown. We can observe a clear improvement in the results of the PUFS+ framewworks in contrast to the ones achieved by the PUFS framework.

As shown in Table 5.2, The PUFS framework averages 31.6018 seconds to find a solution, whereas the PUFS+#MG framework averages 23.3657 seconds and the PUFS+#SG framework 15.4507 seconds. It is important to note that the difference between the average times between the PUFS framework and the PUFS+ frameworks is smaller than expected since the PUFS framework was only able to solve 2

out of the 35 most complex benchmarks. The average runtime takes into account all solutions found and, since the PUFS+ frameworks were able to find more solutions to more complex benchmarks, the average time reaches closer to the PUFS framework.

The precision of the PUFS framework showed to be the worst with 81.81%. In contrast, both PUFS+ frameworks have similar precision around 87%, showing a similar percentage of intended solutions in relation to non intended ones. The recall metric is another significant difference between the original PUFS framework and the PUFS+ frameworks since it shows us that the PUFS framework is only able to find 40.45% intended solutions, in contrast to the 89.53% achieved by the PUFS+#MG framework and the 90.59% achieved by the PUFS+#SG 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 correspond to edge cases. An example of an edge case with our synthesizer is the operations *greater than* and *greater or equal than* for which some examples did not specify properly the edge case that would differentiate the two operations. Another edge case is with the operations *Trim*, *TrimStart* and *TrimEnd*, where, similarly to the previous edge case, some examples allow one or more of these operations to satisfy the specification.

We can conclude that the PUFS+ frameworks achieve by far the best performances in every evaluated metric in comparison to the original PUFS framework. Between the two encodings, the PUFS+#SG framework showed to be overall more efficient and precise than the PUFS+#MG framework and, henceforth, will be the one used to build upon with new features. The difference between the PUFS+ framework encodings is only expected to be bigger the more complex the solutions are.

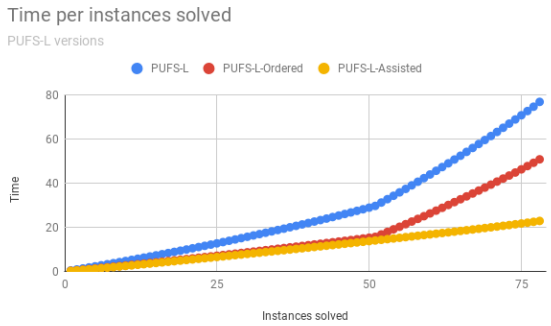
5.3.2 PUFS-L framework

In this section we will analyse the different versions of the PUFS-L framework described in section 4.2. The first version is PUFS-L, where the implementation is simply the addition of a new node *ExecuteAction* and the corresponding operations to the DSL (seen in Tables A.4 and A.5). The second version is the PUFS-L-Ordered framework, which adds a more intelligent way of enumerating through the sketches. Finally, we have the PUFS-L-Assisted framework that adds-on the PUFS-L-Ordered framework by allowing the user to give assistance in the specification.

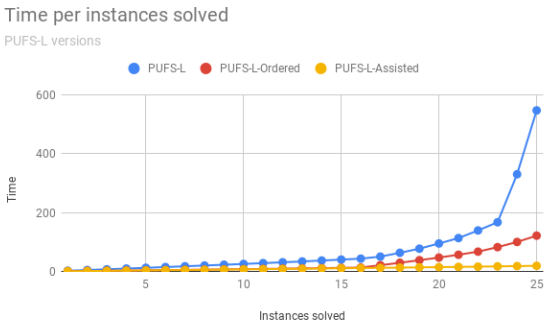
The three versions will be tested with list manipulation benchmarks in section 5.3.2, assignment and conditional benchmarks in section 5.3.2 and assignment, conditional and list manipulation benchmarks in section 5.3.2. With the first set of benchmarks we will have an idea of how the framework behaves with benchmarks that require only list manipulation nodes. The second set of benchmarks will allows us to observe the impact on the framework when we added the new feature, because we will compare the results in the PUFS-L framework in contrast to the PUFS framework. Finally, the last set of benchmarks will give us an idea of how the synthesizer behaves with more complex benchmarks.

List manipulation benchmarks

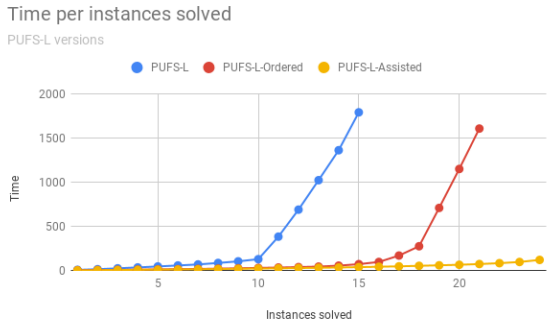
The results for the list manipulation benchmarks can be observed in Figure 5.7, Table 5.4 and Table 5.5.



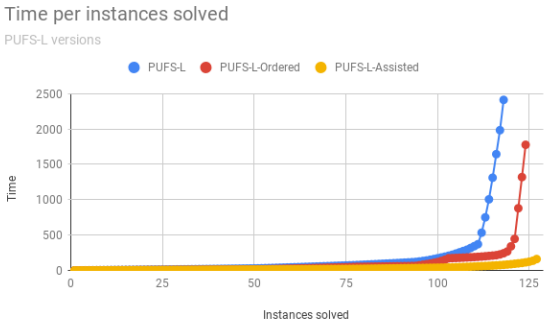
(a) Examples of 1 node



(b) Examples of 2 nodes



(c) Examples of 3 nodes



(d) Examples of 1, 2 and 3 nodes

Figure 5.7: PUFS-L framework versions: list manipulation benchmarks

The same pattern can be observed throughout the benchmarks in Figure 5.7, from 1 node to more complex ones of 3 nodes. The PUFS-L framework has considerably worse results than the other frameworks in every benchmark, ending with a runtime average of 20.4471 seconds. In contrast, the PUFS-L-Ordered framework has a total average of 14.3375 and the PUFS-L-Assisted of 1.2691 seconds for all the benchmarks. The difference between the PUFS-L and PUFS-L-Ordered framework can be attributed to the sketch enumeration process since the PUFS-L version enumerates through every possible sketch independent on the input and output types, whereas the PUFS-L-Ordered framework, before the enumeration, prunes and orders the sketches to remove redundant attempts and maximize the efficiency.

The difference in the average runtime between the PUFS-L-Ordered and PUFS-L-Assisted, as observed in Figure 5.7, is not visible in every benchmark. The only benchmarks where PUFS-L-Ordered has more difficulty are ones that require types CmpLambda and OpLmabda. This is expected since PUFS-L-Ordered does have any assistance from the user, which means it not only needs to have an extra node to create the operation CmpLambda or OpLambda, 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. The difference in the average time is drastic, specially in 3 node benchmarks, where the PUFS-L-Ordered framework averages 76.4571 seconds in contrast to the 4.9667 seconds it takes the PUFS-L-Assisted.

In regards to precision, in 1 node benchmarks all of the frameworks have the value of 1, which is expected since solutions of a single operation do not create any ambiguity with a DSL composed

	PUFS-L			PUFS-L-Ordered			PUFS-L-Assisted		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.9841	1	1	0.6506	1	1	0.2931	1	1
2-nodes	21.8692	0.9259	1	4.86	0.963	1	0.7648	0.963	1
3-nodes	119.2847	0.7143	0.5263	76.4571	0.7619	0.8421	4.9667	0.9583	1
all nodes	20.4471	0.9496	0.9262	14.3375	0.9524	0.9756	1.2691	0.9845	1

Table 5.4: Evaluation metrics for the PUFS-L framework versions: list manipulation benchmarks

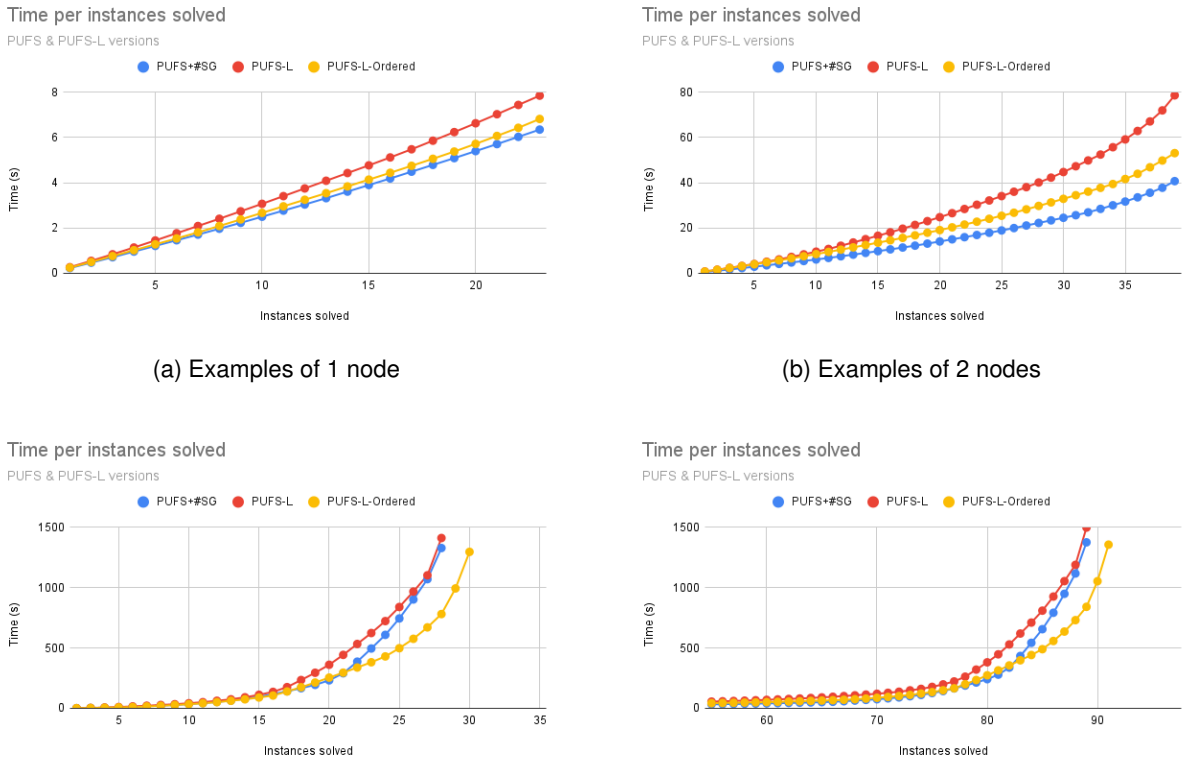
	# Benchmarks	PUFS-L (s)	PUFS-L-Ordered (s)	PUFS-L-Assisted (s)
1-node	78	76.76	50.75	22.86
2-nodes	22	527.1	111.98	16.93
3-nodes	10	684.66	30.38	42.96
all nodes	110	1288.52	193.11	82.75

Table 5.5: Total runtime of same benchmarks for the PUFS-L frameworks: list manipulation benchmarks

of distinct operations. However, with 2 node and 3 node benchmarks the precision worsens. The performance in the precision metric improves when the synthesizer has less possible attempts when we consider an ambiguous specification, because the more different attempts a synthesizer can attempt, the more likely it will find multiple programs that satisfy said specification. Thus, the results for the PUFS-L framework, reaching 0.7143 for 3 node benchmarks, in contrast to the PUFS-L-Assisted framework, which has the highest precision values with 0.9583 for the 3 node benchmarks, are expected.

The recall metric of the frameworks remains at 1 for 1 node and 2 node benchmarks, which tells us that the frameworks were able to iterate through all of the attempts for depth 2. However, in 3 node benchmarks, PUFS-L is only able to find solutions to 52.63% of the benchmarks. As established before in section 4.2.1, the PUFS-L framework needs to iterate through a total of 12 sketches in depth 3. On the other hand, PUFS-L-Ordered has only 5 possible sketches and was able to have a recall of 84.21%. Since this was the only change between frameworks, we can observe the impact on the performance when there are more sketches to enumerate through. The PUFS-L-Assisted framework remained with a recall of 1 for the 3 node benchmarks, which is attributed to the reduced search space due to the assistance of the user, and the lack of need of an extra node to find the operation to use with operations such as *ListMap* or *ListFilter*.

In Table 5.5 we are able to observe the total runtime taken by each framework on the same benchmarks where the solution was correct. As expected and shown throughout the analysis, the PUFS-L framework has significantly worse results throughout the different complexities of the benchmarks, having a total of 1288.52 seconds on all of the same benchmarks. Between the PUFS-L-Ordered frame-



(a) Examples of 1 node

(b) Examples of 2 nodes

(c) Examples of 3 nodes

(d) Examples of 1, 2 and 3 nodes

Figure 5.8: PUFS-L framework versions: assignment and conditional benchmarks

work and the PUFS-L-Assisted, the results are also as expected, with the PUFS-L-Ordered framework improving significantly compared to the PUFS-L framework ending with 193.11 seconds on all of the same benchmarks, and the PUFS-L-Assisted framework ending with 82.75 seconds.

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 correspond to edge cases or, in the case of list manipulation, the confusion between the operations *ListAppend* and *ListInsert*. The former corresponds to the cases where the examples satisfy both the operation $<$ and $<=$ or $>$ and $>=$, which can be resolved by making sure these edge cases are in the examples of the specification. The latter corresponds to the edge case of the operation *ListInsert*, which, when provided with an index that is higher than its size, it functions as a *ListAppend* by inserting the element at the end of the list. Thus, for a benchmark where the solution is *ListAppend*, when the element to be inserted is much higher than the size of the list, a correct solution for the synthesizer is *ListInsert* with the index as the element to be inserted. However, this solution becomes incorrect when the size of the list surpasses the index in the operation *ListInsert*.

Assignment and conditional benchmarks

The results for the assignment and conditional benchmarks can be observed in Figure 5.8, Table 5.6 and Table 5.7. The frameworks evaluated are the PUFS+#SG framework, the PUFS-L framework and the PUFS-L-Ordered framework. The PUFS+#SG framework is used to observe the impact on the

	PUFS+#SG			PUFS-L			PUFS-L-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.2781	1	1	0.3413	1	1	0.2965	1	1
2-nodes	1.0444	0.8205	1	2.0164	0.7949	1	1.3618	0.7692	1
3-nodes	49.1863	0.8519	0.7419	52.2551	0.7407	0.7143	44.6897	0.8966	0.8387
all nodes	15.4507	0.8652	0.9059	16.8245	0.8202	0.9012	14.9003	0.8539	0.9268

Table 5.6: Evaluation metrics for the PUFS-L framework versions: assignment and conditional benchmarks

	# Benchmarks	PUFS+#SG (s)	PUFS-L (s)	PUFS-L-Ordered (s)
1-node	23	6.35	7.85	6.82
2-nodes	28	32.5	59.23	41.42
3-nodes	19	1038.09	1225.48	1038.09
all nodes	63	1338.15	1677.18	888.64

Table 5.7: Total runtime of same benchmarks for the PUFS-L frameworks: assignment and conditional benchmarks

PUFS-L frameworks when running assignment and conditional benchmarks. The framework PUFS-L-Assisted is not evaluated with these benchmarks, because the user assistance feature is never used with operations that are not list manipulations. Thus, the PUFS-L-Ordered and PUFS-L-Assisted framework are equivalent for the assignment and conditional benchmarks. Note that, for these benchmarks, the pruned framework PUFS-L-Ordered framework is able to only enumerate through sketches with nodes of type *Assign* and *If*. Hence, the number of enumerated nodes, after the filtering, becomes the same for the PUFS and PUFS-L-Ordered frameworks.

As we can observe in Figure 5.8a and Table 5.6, with 1 node and 2 node benchmarks, the PUFS+#SG framework is able to have an average runtime of 0.2781 seconds and 1.0444 seconds, respectively, outperforming the other two frameworks. Between the list manipulation frameworks, PUFS-L framework proved to have slightly worse results with 0.3413 seconds for 1 node benchmarks and 2.0164 seconds for 2 node benchmarks, in contrast to 0.2965 seconds and 1.3618 seconds it took the PUFS-L-Ordered framework. This can be attributed to the unnecessary enumeration of sketches the PUFS-L framework performs, such as sketches with the node type *ExecuteAction*, which do not make sense when the input nor the output have elements of type *List*. For 1 node benchmarks, both the precision and recall metrics stayed at the value 1 for every framework as expected due to the simplicity of the benchmarks. However, for 2 node benchmarks, the precision of the frameworks dropped. The reasoning for the lower precision is the same as the one discussed for the PUFS frameworks: edge case scenarios where the specification does not differenciate between operations such as *trim* and *trimStart* or between operations such

	PUFS-L			PUFS-L-Ordered			PUFS-L-Assisted		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
2-nodes	2.104	0.9	1	2.46	0.95	1	1.531	0.95	1
3-nodes	76.5064	1	0.6875	46.4227	0.8636	0.6552	18.8422	0.9658	1

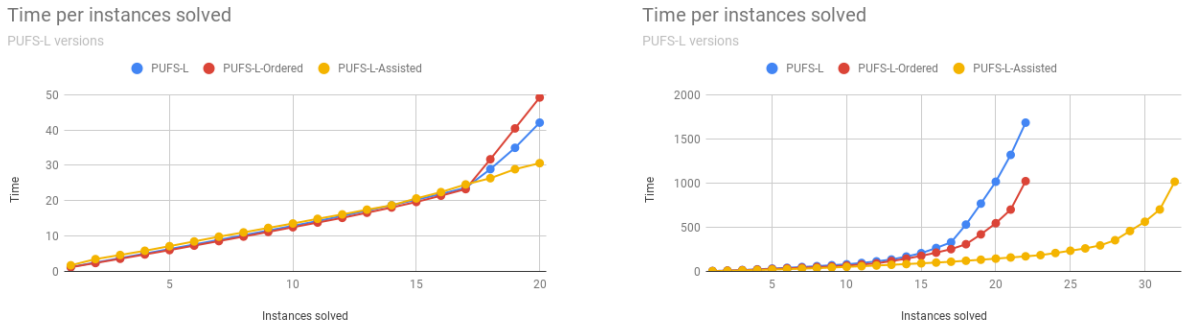
Table 5.8: Evaluation metrics for the PUFS-L framework versions: assignment, conditional and list manipulation benchmarks

as $>$ and $>=$.

For 3 node benchmarks, the PUFS-L framework continues to show the worst results in every metric. However, the PUFS-L-Ordered framework catches up to the PUFS+#SG framework, ending with a better runtime average of 44.6897 seconds in contrast to the 49.1863 seconds. The precision and recall also improve compared to the PUFS+#SG framework. With 3 node benchmarks is when we start to see every framework having difficulties in solving some benchmarks. The best precision is by far the PUFS-L-Ordered framework with 89.66% and the recall was also better than the PUFS+#SG framework with 92.68% in contrast to 90.59%.

In Table 5.7 we can observe the difference in the total runtime spent on the same benchmarks for the PUFS+#SG, PUFS-L and PUFS-L-Ordered framework. The results previously presented and discussed are consolidated by these values since the PUFS-L framework throughout the complexity of benchmarks continued to have the worst results ending with a total of 1677.18 seconds for the same benchmarks. For 1 node and 2 node complexities, the PUFS framework had a better total runtime than the PUFS-L-Assisted framework. Finally, for the 3 node benchmarks, the PUFS-L-Assisted framework surpassed the PUFS+#SG framework ending with 888.64 seconds in contrast to the 1338.15 seconds, which is consistent with the values observed in Figure 5.8a.

All in all, the PUFS-L framework clearly has worse results than the PUFS-L-Ordered framework and will not be considered. The PUFS+#SG framework was able to correctly solve 77 out of the 97 benchmarks whereas the PUFS-L-Ordered framework was able to correctly solve 76. However, overall, the PUFS-L-Ordered framework showed slightly better results in the average runtime ending with 14.9 seconds in contrast to the 15.45 seconds for the PUFS+#SG framework. The reasoning for the similar performance despite one framework having an additional feature is that both frameworks have the exact same DSL since the PUFS-L-Ordered framework is able to notice that sketches with list manipulation, for these benchmarks, will never work. Hence, it not only removes the sketches from the enumerations process, but also the list manipulation operations from the DSL. The only difference between the frameworks is the order in which the SMT solver provides candidate solutions. We can conclude that the impact of having the list manipulation feature is minimal and, thus, is worth pursuing.



(a) Examples with 2 nodes

(b) Examples with 3 nodes

Figure 5.9: PUFS-L framework versions: assignment, conditional and list manipulation benchmarks

	# Benchmarks	PUFS-L (s)	PUFS-L-Ordered (s)	PUFS-L-Assisted (s)
2-nodes	18	38.88	45.98	27.15
3-nodes	18	566.34	713.05	627.97

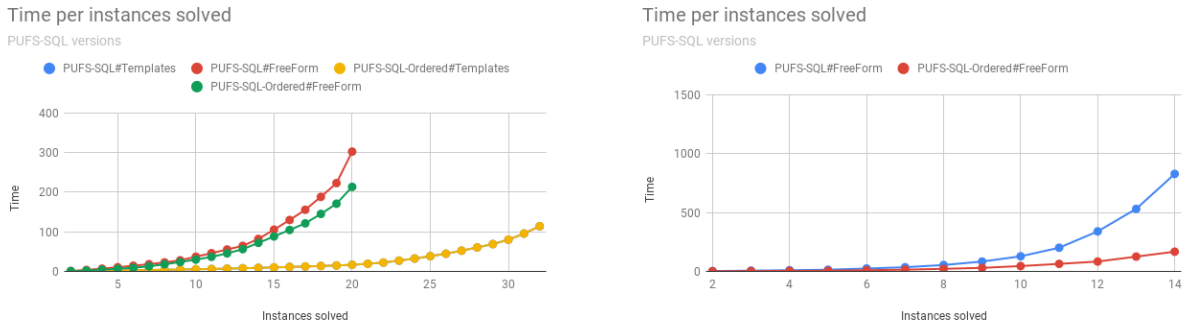
Table 5.9: Total runtime of same benchmarks for the PUFS-L frameworks: assignment, conditional and list manipulation benchmarks

Assignment, conditional and list manipulation benchmarks

The results for the assignment, conditional and list manipulation benchmarks can be observed in Figure 5.9, Table 5.8 and Table 5.9.

As we can observe in Figure 5.9a, for most of the benchmarks the three frameworks have a similar performance. The difference can only be observed for benchmarks that use the operations *CmpLambda* or *OpLambda*, which is expected since the PUFS-L and PUFS-L-Ordered framework do not have guidance of the user and require an extra node to successfully find the solution. The PUFS-L-Assisted framework, with the assistance of the user, can maintain the average runtime at 1.531 seconds in contrast to the 2.104 seconds for the PUFS-L framework and 2.46 seconds for the PUFS-L-Ordered framework. The recall maintained at 1 for all frameworks due to the simplicity of the benchmarks and the precision was close to 1 for all benchmarks, with the errors being due to edge case scenarios as explained in the previous benchmarks.

For 3 node benchmarks, as we can observe in Figure 5.9b, we start to see a major difference between the frameworks. As expected, the PUFS-L framework has the least efficient results due to the simplicity of the implementation, averaging 76.5064 seconds per benchmark. However, the PUFS-L-Ordered framework, even though the efficiency had an improvement to 46.4227 seconds due to the pruning and ordering of sketches, the precision and recall suffered. The cause for the lower precision values are the edge cases, which are attributed to the ambiguity of the specification. Despite the slight worse results, the PUFS-L-Ordered framework continues to be considered better than the PUFS-L framework since the average runtime was reduced to almost half and the lower precision can be solved with a more cautious specification. The PUFS-L-Assisted framework, similarly to the 2 node bench-



(a) Examples with template benchmarks

(b) Examples with free-form benchmarks

Figure 5.10: PUFS-SQL framework versions: data aggregation benchmarks

marks, continues to show better results than any other framework, with an average of 18.8422 seconds, a recall of 1 and a precision of 0.9658. Due to the natural exponential complexity of the problem, 3 node solutions are already significantly more efficient than 4 node solutions. The PUFS-L-Assisted framework was able to have much better results because both PUFS-L and PUFS-L-Ordered framework need an extra node for benchmarks that use operations such as *ListFilter* or *ListMap* becoming a 4 node benchmark.

In Table 5.9, we can observe the total runtime spent by each framework on the same benchmarks. For 2 node benchmarks, the PUFS-L framework spent a total of 38.88 seconds on the same 18 benchmarks, whereas the PUFS-L-Ordered framework spent 45.98 seconds, which is similar to the average runtimes observed in Table 5.8 and can be attributed to the added overhead the PUFS-L-Ordered framework has. The PUFS-L-Assisted framework spent the least amount of runtime with a total of 27.15 seconds. For 3 node benchmarks, the results differ from the previous conclusions with the PUFS-L framework being the most efficient, followed by the PUFS-L-Assisted framework and, finally, the PUFS-L-Ordered framework. This occurs in these benchmarks, because all of the more complex benchmarks that make use of the operations *ListMap* or *ListFilter* where the PUFS-L-Assisted framework excels at are not included in these benchmarks since a solution was not found by at least one of the remaining frameworks.

All in all, the PUFS-L-Assisted framework is the clear best framework out of all versions by allowing the user to guide the synthesizer and will be the one used to build upon with new features. Note that the PUFS-L-Assited framework works whether the user provides assistance or not, in which case the results should be similar to the ones seen in PUFS-L-Ordered framework.

The addition of the list manipulation capability, creating the PUFS-L framework, does not seem to have a negative effect on the assignment and conditional benchmarks due to the pruning and ordering of sketches. Hence, since the new framework is able to solve the same problems with as much if not higher efficiency than the PUFS+ framework and has an extra feature, it becomes the best of both frameworks.

	PUFS-SQL#FreeForm			PUFS-SQL#Templates			PUFS-SQL-Ordered#FreeForm			PUFS-SQL-Ordered#Templates		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
Templates	15.91	1	0.5938	5.3072	1	1	11.2237	1	0.5938	5.2422	1	1
Free-form	89.46	1	1	–	–	–	38.16	1	1	–	–	–

Table 5.10: Evaluation metrics for the PUFS-SQL framework versions: data aggregation benchmarks

	# Benchmarks	PUFS-SQL#FreeForm (s)	PUFS-SQL#Templates (s)	PUFS-SQL-Ordered#FreeForm (s)	PUFS-SQL-Ordered#Templates (s)
Templates	19	302.27	26.77	206.33	26.77
Free-form	14	1252.37	–	534.22	–

Table 5.11: Total runtime of same benchmarks for the PUFS-SQL frameworks: data aggregation benchmarks

5.3.3 PUFS-SQL framework

In this section we will analyse the different versions of the PUFS-SQL framework described in section 4.3. The first version is PUFS-SQL#FreeForm, where the implementation is the addition of a new node *DataSet* and the integration with the CUBES framework with the corresponding operations to the DSL (seen in Tables A.6). The second version is the PUFS-SQL#Templates framework, which makes use of a set of most used templates from the OutSystems platform to simplify the synthesizing of the benchmarks. Finally, we have the PUFS-SQL-Ordered#FreeForm and PUFS-SQL-Ordered#Template frameworks, which add a more intelligent way of enumerating through the sketches and use the respective encoding of free-form or templates.

The four versions will be tested with data aggregation benchmarks in section 5.3.3, assignment and conditional benchmarks in section 5.3.3 and assignment, conditional and data aggregation benchmarks in section 5.3.3. With the first set of benchmarks we will have an idea of how the framework behaves with benchmarks that require only data aggregation nodes. The second set of benchmarks will allows us to observe the impact on the framework when we added the new feature, because we will compare the results in the PUFS-SQL framework in contrast to the PUFS framework. Finally, the last set of benchmarks will give us an idea of how the synthesizer behaves with more complex benchmarks.

Data aggregation benchmarks

The results for the data aggregation benchmarks can be observed in Figure 5.10, Table 5.10 and Table 5.11. The benchmarks are divided into 2 different categories: template benchmarks, which have a solution using the templates described in section 4.3.1; and freeform benchmarks, which do not have a solution using templates, but only with free-form SQL queries.

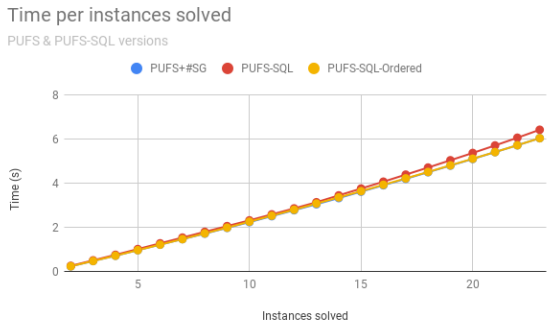
For the template benchmarks we can immediately see a clear difference between running the framework with the template configuration versus the free-form configuration. The PUFS-SQL#Templates framework, as shown in Table 5.10, averages 5.3072 seconds in solving 32 instances in contrast to the 15.91 seconds it takes for the PUFS-SQL#FreeForm to find a solution for only 19 benchmarks. The later framework is unable to find solutions to more complex queries or bigger tables, ending with a recall of

only 59.38%. Thus, the difference in the average time is even more impactful, knowing that the PUFS-SQL#Templates framework was able to find solutions to all benchmarks. The results are expected since the templates of the PUFS-SQL#Templates framework take a complex function and find a solution in a single node, whereas the PUFS-SQL#FreeForm framework simply uses the operations of the CUBES framework and can require from 1 node to 3 nodes to find the same solution. Furthermore, in Table 5.11 we can see the total runtime spent by each framework on the same benchmarks and, in 19 benchmarks, the PUFS-SQL#Templates framework took a total of 26.77 seconds to find the solutions in contrast to the 302.27 seconds spent by the PUFS-SQL#FreeForm framework. The precision for both frameworks remains at 1, which means there was no ambiguous solutions for the selected benchmarks.

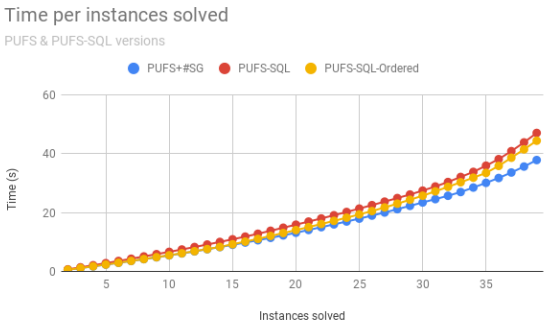
We can observe the impact of the filtering and ordering of sketches by first comparing the results of the PUFS-SQL#Templates framework with the PUFS-SQL-Ordered#Templates framework and then the results of the PUFS-SQL#FreeForm framework with the PUFS-SQL-Ordered#FreeForm framework. For the first comparison, the difference between frameworks is not visible in Figure 5.10a. The change is not significant, which is expected since all of the benchmarks have a solution of a single node. The pruning and ordering of sketches should only show with benchmarks that require more than a single node to find a solution. With the free-form configuration, the difference is already visible which goes hand in hand with the solution requiring more than a single node to reach a solution. The average runtime with the pruning and ordering changed from 15.91 seconds to 11.2237 seconds, as shown in Table 5.10. The total runtime spent by each framework, as shown in Table 5.11, also demonstrate the difference with the PUFS-SQL#Freeform framework having a total of 302.27 seconds in contrast to the 206.33 of the PUFS-SQL-Ordered#Freeform framework.

For the free-form benchmarks, the only frameworks ran were the PUFS-SQL#FreeForm and PUFS-SQL-Ordered#FreeForm, because the template frameworks could never find a solution since these benchmarks specifically target the areas that templates can not be used. As observed in Figure 5.10b, the PUFS-SQL-Ordered#FreeForm framework performs better than the PUFS-SQL#FreeForm framework in their runtime averaging 38.16 seconds in contrast to the 89.46 seconds. In the total time spent on the same frameworks, which can be seen in 5.11, the same difference can be observed with the PUFS-SQL-Ordered#FreeForm spending a total of 534.22 seconds in 14 benchmarks in contrast to the 1252.37 seconds spent by the PUFS-SQL#FreeForm framework. The precision maintained at 1 for both frameworks, which means there was no ambiguity in the selected benchmarks. The recall also maintained at 1, because free-form benchmarks with more than 2 node operations start to be too complex for the frameworks to provide solutions within the pre-determined time limit and, therefore, could not be evaluated.

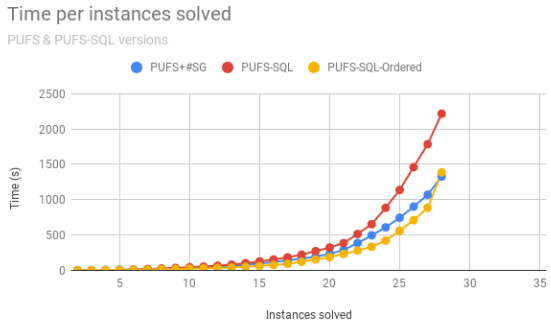
All in all, the best performing frameworks follow the template method and, with the pruning and ordering of sketches, can further improve its performance. Due to the drastic difference in performance and the representation of majority of benchmarks, the PUFS-SQL-Ordered#Templates framework will be the version used in the remaining evaluations. However, the configuration to allow free-form queries will always be present in the framework in case the templates are not enough to find a solution.



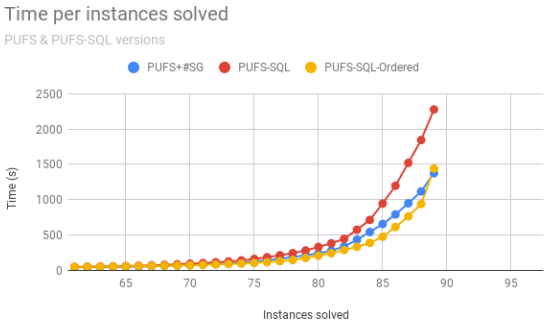
(a) Examples of 1 node



(b) Examples of 2 nodes



(c) Examples of 3 nodes



(d) Examples of 1, 2 and 3 nodes

Figure 5.11: PUFS-SQL framework versions: assignment and conditional benchmarks

Assignment and conditional benchmarks

The results for the assign and conditional benchmarks can be observed in Figure 5.11, Table 5.12 and Table 5.13. The benchmarks were ran with the PUFS+, PUFS-SQL and PUFS-SQL-Ordered framework, using the templates framework since the performance compared to the free-form framework is significantly better in the metrics evaluated. Note that, for these benchmarks, the pruned framework PUFS-SQL-Ordered framework is able to only enumerate through sketches with nodes of type *Assign* and *If*. Hence, the number of enumerated nodes, after the filtering, becomes the same for the PUFS+ and PUFS-SQL-Ordered frameworks.

As observed in Figure 5.11, the results for all three frameworks are very similar throughout the different complexities, with the 3 node benchmarks distinguishing more clearly the differences between the frameworks in terms of performance. In Table 5.12, we see that the average time spent on the benchmarks by the PUFS+ framework in 1 node and 2 node benchmarks averaged 0.2781 seconds and 1.0444 seconds, respectively, whereas the PUFS-SQL-Ordered framework had similar but slightly better results averaging 0.277 seconds and 1.22 seconds, respectively. The PUFS-SQL framework averaged 0.2983 seconds and 1.3551 seconds, respectively, which is slightly worse than the other frameworks. The results are consolidated by the values in Table 5.13, where the total time spent by each framework on the same benchmarks is presented. The PUFS-SQL framework and PUFS-SQL-Ordered framework have the worst results in 1 node and 2 node benchmarks, whereas the PUFS+ framework has the best performance. We can also observe that the difference between the PUFS-SQL and PUFS-SQL-Ordered

	PUFS+#SG			PUFS-SQL			PUFS-SQL-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.2781	1	1	0.2983	1	1	0.2770	1	1
2-nodes	1.0444	0.8205	1	1.3551	0.8205	1	1.22	0.8205	1
3-nodes	49.1863	0.8519	0.7419	82.0929	0.7778	0.7241	51.3926	0.8519	0.7419
all nodes	15.4507	0.8652	0.9059	25.5163	0.8539	0.9048	16.1735	0.8736	0.9048

Table 5.12: Evaluation metrics for the PUFS-SQL frameworks: assignment and conditional benchmarks

	# Benchmarks	PUFS+#SG (s)	PUFS-SQL (s)	PUFS-SQL-Ordered (s)
1-node	23	6.35	6.86	6.37
2-nodes	28	33.61	41.55	35.72
3-nodes	18	966.15	1880.36	1147.41
all nodes	69	1006.11	1928.77	1189.5

Table 5.13: Total runtime of same benchmarks for the PUFS-SQL frameworks: assignment and conditional benchmarks

is more significant than the difference between the PUFS-SQL-Ordered and PUFS+ frameworks.

For 1 node and 2 node benchmarks the precision and recall of the three frameworks is exaclty the same. The precision remained at 100% for 1 node benchmarks, which is expected since more simple benchmarks do not have as much ambiguity. However, for 2 node benchmarks, the precision of all three frameworks lowered to 82.05%. As explained in the previous analysis of assign and conditional benchmarks, the lower precision is attributed to edge cases where the specification does not properly differ edge cases to determine if, for example, the solution should be *greater than* or *greater or equal than*.

With 3 node benchmarks, despite the Figure 5.11c showing that the PUFS+ framework seems slightly worse, the average runtime continued to be lower than the PUFS-SQL-Ordered framework, ending with 49.1863 seconds in contrast to 51.3926 seconds. The difference between the PUFS-SQL and PUFS-SQL-Ordered frameworks becomes more clear, with the PUFS-SQL averaging 82.0929 seconds to find a solution. The difference between the frameworks is expected, since the more complex a benchmark is the more significant the pruning and ordering of sketches should be. The precision and recall of the PUFS-SQL framework lowered to around 77% and 72%, respectively. The PUFS+ and PUFS-SQL-Ordered frameworks were able to maintain a high percentage of precision of 85.19%. The recall, however, took a toll lowering from 100% to 74.19%. In Table 5.13, we consolidate the analysis by observing the total time spent by each framework on the same benchmarks. The PUFS-SQL framework spent 1880.36 seconds on 18 benchmarks, whereas the PUFS-SQL-Ordered framework spent 1147.41 seconds and the PUFS+ framework 966.15 seconds.

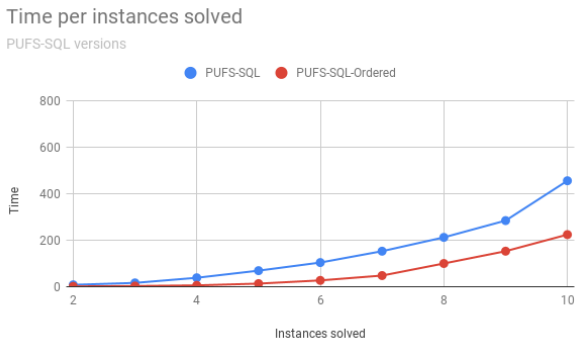


Figure 5.12: PUFS-SQL framework versions: assignment, conditional and data aggregation benchmarks

	PUFS-SQL			PUFS-SQL-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall
3-nodes	69.065	1	1	38.07	1	1

Table 5.14: Evaluation metrics for the PUFS-SQL frameworks: assignment, conditional and data aggregation benchmarks

All in all, the addition of the data aggregation feature slightly affects the performance of assign and conditional benchmarks when using the most advanced framework PUFS-SQL-Ordered. However, considering the addition of the new feature, the slight impact seems like a worth while trade-off.

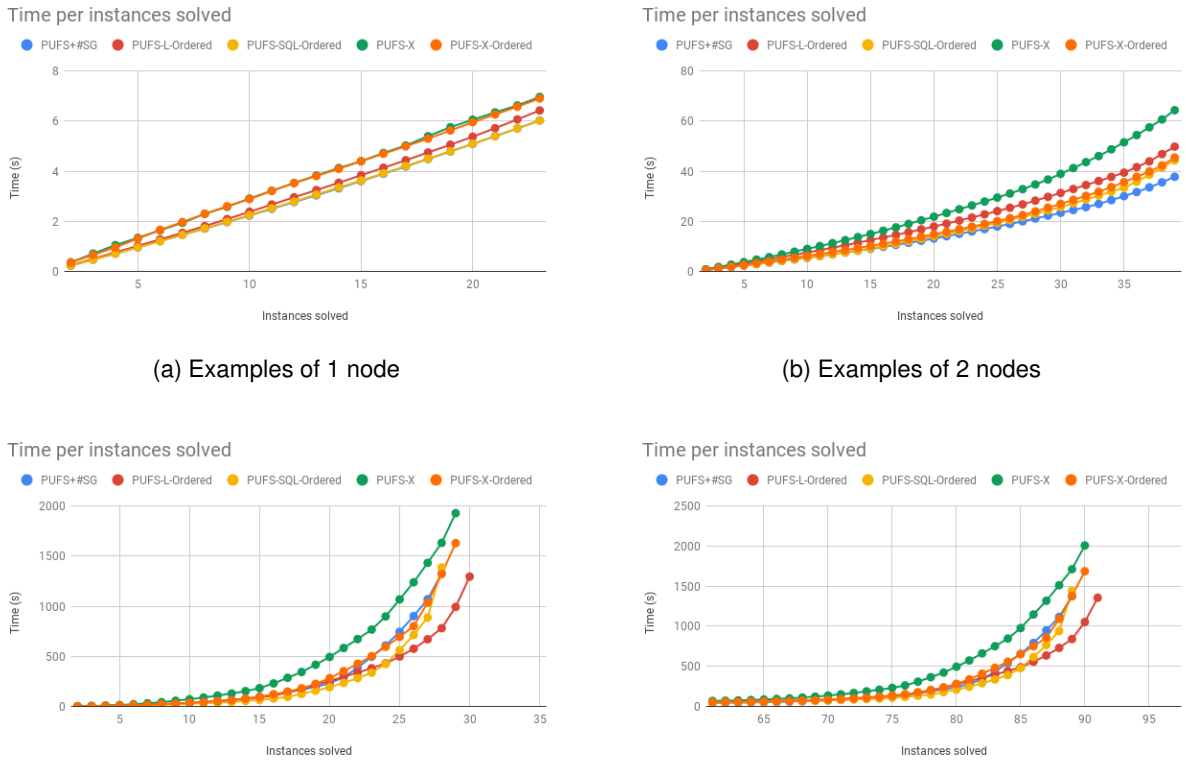
Assignment, conditional and data aggregation benchmarks

The results for the assign, conditional and data aggregation benchmarks can be observed in Figure 5.12, Table 5.14 and Table 5.15. The benchmarks were ran with the PUFS-SQL and PUFS-SQL-Ordered framework, using the templates framework since the performance compared to the free-form framework is significantly better in the metrics evaluated for only data aggregation benchmarks. As explained in detail in section 5.1.5, the benchmarks consist in only queries and if conditions, which implies that the number of minimum nodes needed is 3.

As we can observe in Figure 5.12, the PUFS-SQL-Ordered framework is more efficient than the PUFS-SQL framework, which is expected due to the filtering and ordering of the sketches. The average runtime for the PUFS-SQL framework is 69.065 seconds in contrast to the 38.07 seconds in average for the PUFS-SQL-Ordered framework. The same can be observed in Table 5.15, where the PUFS-

	# Benchmarks	PUFS-SQL (s)	PUFS-SQL-Ordered (s)
3-nodes	10	690.65	380.7

Table 5.15: Total runtime of same benchmarks for the PUFS-SQL frameworks: assignment, conditional and data aggregation benchmarks



(a) Examples of 1 node

(b) Examples of 2 nodes

(c) Examples of 3 nodes

(d) Examples of 1, 2 and 3 nodes

Figure 5.13: PUFS-X framework versions: assignment and conditional benchmarks

SQL framework spends a total of 690.65 seconds on the same benchmarks as the PUFS-SQL-Ordered framework, which spent a total of 380.7 seconds.

The precision and recall is the same for both frameworks, being both 100%, which shows us that there was no ambiguity in the specification.

All in all, the PUFS-SQL framework's best configurations are with the template method and with the pruning and ordering of sketches. With these configurations, the framework is able to solve problems with SQL queries and the same types of problems the PUFS+ framework can with a small impact on the performance. Hence, considering the addition of the new feature and that the difference in performance compared with the PUFS+ framework is small, the framework is worth pursuing.

5.3.4 PUFS-X framework

In this section we will analyse the different versions of the PUFS-X framework described in section 4.4. The first version is the PUFS-X framework, which simply joins the PUFS, PUFS-L and PUFS-SQL framework into a single one. The second version is the PUFS-X-Ordered framework, which add a more intelligent way of enumerating through the sketches.

The two versions will be tested with every benchmark set used so far in order to observe the impact when using a framework that has all of the capabilities proposed. Then, benchmarks with list manipulation and data aggregation capabilities and benchmarks with all three capabilities are run in order to observe the performance when using more complex benchmarks.

	PUFS+#SG			PUFS-L-Ordered			PUFS-SQL-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.2781	1	1	0.2965	1	1	0.277	1	1
2-nodes	1.0444	0.8205	1	1.3618	0.7692	1	1.22	0.8205	1
3-nodes	49.1863	0.8519	0.7419	44.6897	0.8966	0.8387	51.3926	0.8519	0.7419
all nodes	15.4507	0.8652	0.9059	14.9003	0.8539	0.9268	16.1735	0.8736	0.9048

Table 5.16: Evaluation metrics for the PUFS, PUFS-L and PUFS-SQL framework versions: assignment and conditional benchmarks

	PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.3165	1	1	0.3135	1	1
2-nodes	1.8415	0.8462	1	1.2590	0.8462	1
3-nodes	68.9221	0.8214	0.7667	58.2321	0.8571	0.7742
all nodes	22.3213	0.8778	0.9186	18.7423	0.8889	0.9195

Table 5.17: Evaluation metrics for the PUFS-X frameworks: assignment and conditional benchmarks

Assignment and conditional benchmarks

The results for the assign and conditional benchmarks can be observed in Figure 5.13, Table 5.16, Table 5.17 and Table 5.18. The results presented are from the frameworks PUFS+#SG, PUFS-L-Ordered and PUFS-SQL-Ordered#Templates, which have already been presented and analysed, but will be reused here to facilitate the comparison between frameworks. Then, we have the new results obtained by the PUFS-X and PUFS-X-Ordered frameworks. Note that, for assignment and conditional benchmarks, the frameworks that are pruned and ordered are able to only enumerate through sketches with nodes of type *Assign* and *If*. Hence, the number of enumerated nodes, after the filtering, becomes the same for every pruned framework.

For 1 node and 2 node benchmarks, the difference between the performance of the frameworks is

	# Benchmarks	PUFS+#SG (s)	PUFS-L-Ordered (s)	PUFS-SQL-Ordered (s)	PUFS-X (s)	PUFS-X-Ordered (s)
1-node	23	6.35	6.82	6.37	7.28	7.21
2-nodes	25	26.25	35.5	32.72	49.03	34.15
3-nodes	16	1006.18	394.48	938.03	1118.31	577.45
all nodes	64	1038.78	436.8	977.12	1174.62	618.81

Table 5.18: Total runtime of same benchmarks for the all frameworks: assignment and conditional benchmarks

not significant with the PUFS+#SG framework having the best runtime average of 0.2781 seconds for 1 node benchmarks and 1.0444 seconds for 2 node benchmarks. The worst performance is the PUFS-X framework, which is expected since it does not prune or order the sketches. The framework averaged 0.3165 seconds for 1 node benchmarks and 1.8415 seconds for 2 node benchmarks. The precision and recall is similar for all frameworks for both 1 node and 2 node benchmarks, with the precision lowering below 1 in 2 node benchmarks. The reasoning for the lower precision maintains the same as explained in the PUFS frameworks analysis: the cases where the solution found is not the intended one are edge cases not properly differentiated in the specification.

With 3 node benchmarks the results are not as linear because the best performance in the average runtime, surpassing the PUFS+#SG framework, is the PUFS-L-Ordered framework, which was already analysed in section 5.1.3. The PUFS-X framework is the clear worst once again, averaging 68.92 seconds in contrast to the PUFS-L-Ordered framework that averaged 44.6897 seconds. With the pruning and ordering of sketches that the PUFS-X-Ordered framework provides, the difference in the average runtime reduces to 58.2321 seconds. It is important to note that the recall of both the PUFS-X and PUFS-X-Ordered framework are slightly higher than the PUFS+#SG framework, but still lower than the PUFS-L-Ordered framework.

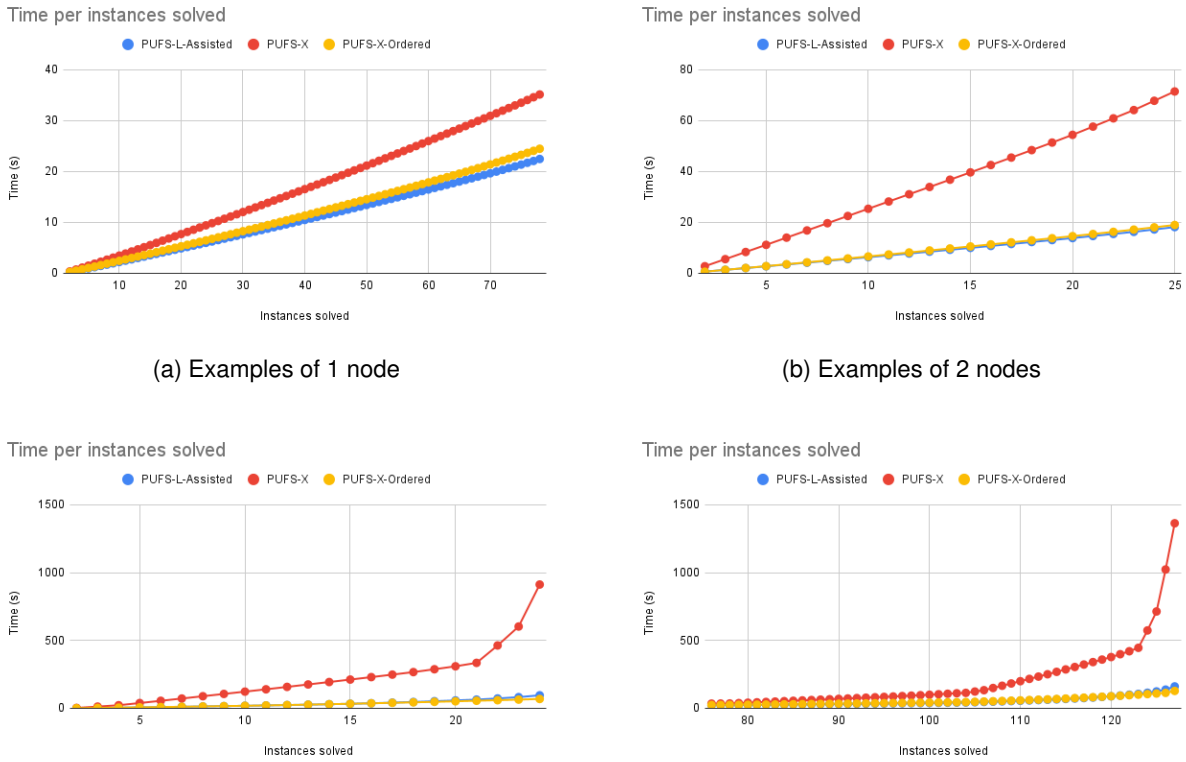
In Table 5.18 we can observe the total time spent by each framework on the same benchmarks. The best performance is the PUFS-L-Ordered framework, followed by the PUFS-X-Ordered framework. It is surprising to not see the PUFS+#SG ahead the PUFS-X-Ordered framework since it does not need to prune the sketches. Knowing that the DSL and interpreter are identical, the only reason for the difference in performance is the order in which the candidate solutions are provided by the SMT solver.

All in all, the PUFS-X-Ordered framework shows worse results in comparison to the PUFS+#SG and PUFS-L-Ordered frameworks in terms of the average runtime. However, the precision and recall, overall, are slightly higher than the other frameworks, reaching a peak of 88.89% of precision and 91.95% of recall. With these metrics in mind, with the ability of synthesizing programs with assignment, conditional, list manipulation and data aggregation, the PUFS-X-Ordered framework loses on efficiency but ultimately allows much more complex programs and, hence, seems to be worth pursuing.

List manipulation benchmarks

The results for the list manipulation benchmarks can be observed in Figure 5.14, Table 5.19 and Table 5.20. The results presented are from the frameworks PUFS-L-Ordered, which have already been presented and analysed, but will be reused here to facilitate the comparison between frameworks. Then, we have the new results obtained by the PUFS-X and PUFS-X-Ordered frameworks. Note that, for list manipulation benchmarks, the frameworks that are pruned and ordered are able to only enumerate through sketches with nodes of type *Assign*, *If* and *ExecuteAction*. Hence, the number of enumerated nodes, after the filtering, becomes the same for the PUFS-L-Ordered and PUFS-X-Ordered frameworks.

As seen in Figure 5.14, for 1 node and 2 node benchmarks, the PUFS-L-Assisted framework performs better than the other frameworks in terms of efficiency. In Table 5.19, we can observe that the average time for 1 node benchmarks is 0.2931 seconds and, for 2 node benchmarks, is 0.7648 seconds.



(a) Examples of 1 node

(b) Examples of 2 nodes

(c) Examples of 3 nodes

(d) Examples of 1, 2 and 3 nodes

Figure 5.14: PUFS-X framework versions: list manipulation benchmarks

As expected, the PUFS-X framework is the clear worst performing framework due to the lack of pruning and ordering of sketches, averaging 0.4574 seconds for 1 node benchmarks and 3.005 seconds for 2 node benchmarks. With the pruning and ordering of sketches, the PUFS-X-Ordered framework is able to reach close to the performance of the PUFS-L-Assisted framework, averaging 0.3194 seconds for 1 node benchmarks and 0.7964 for 2 nodes. The recall maintains at 1 for all benchmarks. However, the precision lowers with 2 node benchmarks, with the PUFS-L-Assisted framework reaching 96.3%, but the PUFS-X framework worsening to 88% and the PUFS-X-Ordered framework to 92%.

For 3 node benchmarks, the PUFS-X-Ordered framework is able to surpass the PUFS-L-Assisted framework, averaging 3.4683 seconds in contrast to the 4.9667 seconds. However, the precision suffered lowering from 95.83% to 87.5%. If we observe Table 5.20, we can see total time spent by all frameworks for the instances that were able to reach the correct solution. Here, we can confirm the previous results of the average times, because for 1 node and 2 node benchmarks, the PUFS-X-Ordered framework is worse than the PUFS-L-Assisted framework, but in 3 node benchmarks it surpasses it.

All in all, the PUFS-X framework is the clear worst performing synthesizer, averaging 10.7345 seconds to solve a benchmark and with the lowest overall precision of 95.28%. The PUFS-X-Ordered framework has a significant increase in its performance, even bettering the PUFS-L-Assisted framework by 0.2 seconds in the total average runtime. In terms of the total precision, the PUFS-X-Assisted framework continued to best the other frameworks with a total of 98.45%. In Table 5.20, we can observe that, for the same benchmarks, the PUFS-X-Ordered framework ends up with a more efficient performance than

	PUFS-L-Assisted			PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
1-node	0.2931	1	1	0.4574	1	1	0.3194	1	1
2-nodes	0.7648	0.963	1	3.005	0.88	1	0.7964	0.92	1
3-nodes	4.9667	0.9583	1	52.1863	0.875	1	3.4683	0.875	1
all nodes	1.2691	0.9845	1	10.7345	0.9528	1	1.0083	0.9606	1

Table 5.19: Evaluation metrics for the PUFS-X frameworks: list manipulation benchmarks

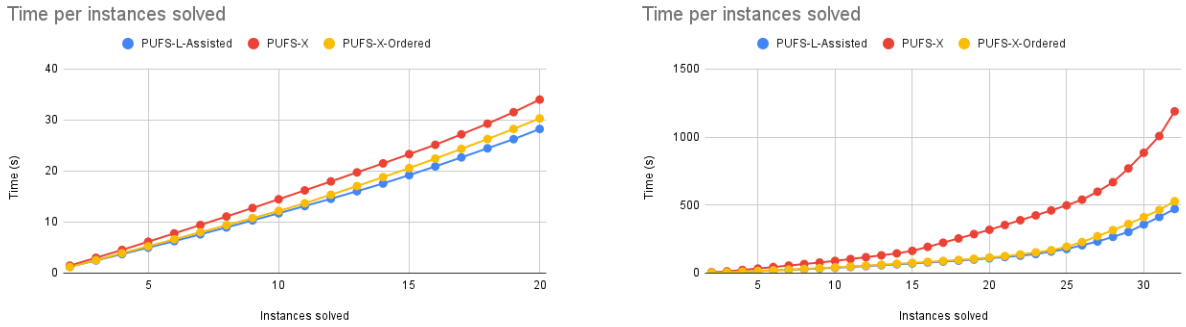
	# Benchmarks	PUFS-L-Ordered (s)	PUFS-X (s)	PUFS-X-Ordered (s)
1-node	78	22.86	35.68	24.91
2-nodes	22	16.66	65.3	17.4
3-nodes	21	97.9	907.89	72.28
all nodes	121	137.42	1008.87	114.59

Table 5.20: Total runtime of same benchmarks for the PUFS-X frameworks: list manipulation benchmarks

the remaining frameworks.

Assignment, conditional and list manipulation benchmarks

The results for the assignment, conditional list manipulation benchmarks can be observed in Figure 5.15, Table 5.21 and Table 5.22. The results presented are from the framework PUFS-L-Ordered, which has already been presented and analysed, but will be reused here to facilitate the comparison between frameworks. Then, we have the new results obtained by the PUFS-X and PUFS-X-Ordered frameworks. Note that, similarly to the list manipulation benchmarks, with these benchmarks, the frameworks that



(a) Examples with 2 nodes

(b) Examples with 3 nodes

Figure 5.15: PUFS-X framework versions: assignment, conditional and list manipulation benchmarks

	PUFS-L-Assisted			PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
2-nodes	1.5318	0.95	1	1.827	1	1	1.6425	1	1
3-nodes	18.8422	0.9658	1	43.6322	0.9658	1	21.7744	0.9658	1

Table 5.21: Evaluation metrics for the PUFS-X frameworks: assignment, conditional and list manipulation benchmarks

are pruned and ordered are able to only enumerate through sketches with nodes of type *Assign*, *If* and *ExecuteAction*. Hence, the number of enumerated nodes, after the filtering, becomes the same for the PUFS-L-Ordered and PUFS-X-Ordered frameworks.

The same pattern can be observed independently of the complexity of the benchmarks. The PUFS-L-Assisted framework performs better than the remaining frameworks and the PUFS-X framework is the clear worst. As in the previous benchmarks, there is no surprise in the performance of the frameworks, since PUFS-X does not prune or order the sketches.

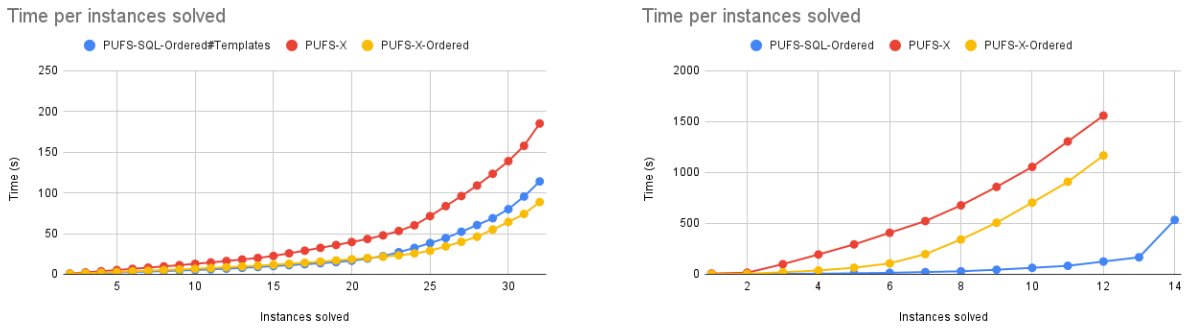
For 2 node benchmarks, the PUFS-L-Assisted averaged 1.5318 seconds in runtime, whereas the PUFS-X-Ordered framework had slight worse average of 1.6425 seconds. The precision of the PUFS-X frameworks was slightly better than the PUFS-L-Assisted framework by being able to find the intended solution to all benchmarks in contrast to the single benchmark the PUFS-L-Assisted was unable to. This benchmark corresponds to the ambiguity analysed and presented in previous types of benchmarks: the equivalence of the *ListInsert* and *ListAppend* functions if the index of insertion is higher than the size of the list. The only reason why the PUFS-X frameworks did not fall for this ambiguity is the order of candidate solutions the SMT solver provides. The recall maintained at 1 for benchmarks, indicating that there was no difficulty in finding a solution within the pre-determined time limit.

For 3 node benchmarks, we can already see the average time increasing quite significantly, exposing once again the exponential complexity of the problem. The PUFS-L-Assisted framework averaged 18.8422 seconds in the runtime whereas the PUFS-X framework averaged 43.6322 seconds and the PUFS-X-Ordered framework 21.7744 seconds. Usually the time difference between frameworks that prune and order sketches and the ones that do not is not double. However, with the PUFS-X frameworks we see this because it has 4 different types of nodes and, if there is no pruning or ordering, for a depth d, we have 4^d sketches to enumerate through. The precision and recall maintained the same throughout the benchmarks, with the precision ending with 96.58% and the recall at 100%.

All in all, there is a difference in performance between the PUFS-L-Assisted and PUFS-X-Ordered framework that is worth noticing. However, the ability to find more complex solutions outweights the slight loss in performance and, therefore, the PUFS-X framework continues to seem to be worth pursuing.

	# Benchmarks	PUFS-L-Assisted (s)	PUFS-X (s)	PUFS-X-Ordered (s)
2-nodes	19	28.25	34.73	30.31
3-nodes	21	97.9	907.89	72.28

Table 5.22: Total runtime of same benchmarks for the PUFS-X frameworks: assignment, conditional and list manipulation benchmarks



(a) Examples with template benchmarks

(b) Examples with free-form benchmarks

Figure 5.16: PUFS-X framework versions: data aggregation benchmarks

Data Aggregation benchmarks

The results for the data aggregation benchmarks can be observed in Figure 5.16, Table 5.23 and Table 5.24. The results presented are from the framework PUFS-SQL-Ordered, which has already been presented and analysed, but will be reused here to facilitate the comparison between frameworks. Then, we have the new results obtained by the PUFS-X and PUFS-X-Ordered frameworks. For the analysis of the template benchmarks, the PUFS-SQL-Ordered framework is ran with the template configuration, which has proven to produce significantly better results than the free-form method. However, for the analysis of the free-form benchmarks, the PUFS-SQL-Ordered framework is ran with the free-form method since these benchmarks represent the percentage of solutions that can never be reached with only templates and, therefore, require a free-form SQL query synthesis.

Note that, with these benchmarks, the PUFS-X-Ordered framework that is pruned and ordered is not able to reduce the types of nodes accepted, because, from an SQL query, every other type of node may be used. The main difference is the guidance, removing sketches that could never work, such as having a node of type *Assign* after a node of type *DataSet*.

For the template benchmarks, as seen in Figure 5.16a, the PUFS-X-Ordered framework was able to have a better performance than the PUFS-SQL-Ordered framework reaching an average of 3.4025 seconds in contrast to the 5.2422 seconds as detailed in Table 5.23. As we can observe, similarly to every other analysed benchmark, the PUFS-X framework is the clear worst framework averaging 6.7778 seconds. This framework showing worse results is expected due to the lack of pruning and ordering of sketches. The PUFS-X-Ordered framework produced better results than the PUFS-SQL-Ordered framework, which could be surprising but we must remember that the template benchmarks have a solution

	PUFS-SQL-Assisted			PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
Templates	5.2422	1	1	6.7778	1	1	3.4025	1	1
Free-form	38.16	1	1	129.8658	0.8571	1	97.2192	0.8571	1

Table 5.23: Evaluation metrics for the PUFS-X frameworks: data aggregation benchmarks

	# Benchmarks	PUFS-SQL-Ordered (s)	PUFS-X (s)	PUFS-X-Ordered (s)
Templates	32	167.75	216.89	108.88
Free-form	12	125.3	1558.75	1166.63

Table 5.24: Total runtime of same benchmarks for the PUFS-X frameworks: data aggregation benchmarks

of a single node, which means that the PUFS-X-Ordered and PUFS-SQL-Ordered frameworks will both attempt first the sketch with the single node *DataSet*. Thus, the difference between the frameworks is minimal. The main difference is the order in which the SMT solver provides solutions, which, for the case of the PUFS-X-Ordered framework made it so the average runtime became lower than the PUFS-SQL-Ordered framework. In Table 5.24, the total time spent by each framework on the same benchmarks is presented, which consolidates the analysis made. The recall and precision maintained at 1 for every framework, which is expected since the template method performs queries using a single node. Ambiguity and difficulty in finding solutions should only occur in more complex sketches, i.e., sketches with a larger number of nodes.

For the free-form benchmarks, the PUFS-SQL-Ordered, as seen in Figure 5.16b, showed the best performance by far. This is expected since when we use the free-form method the sketches may need between 1 node to 3 nodes to find a solution and, as mentioned previously, for benchmarks with data aggregation, the PUFS-X-Ordered framework is forced to enumerate sketches with every type of node. Hence, for 3 node benchmarks for instance, the PUFS-X-Ordered framework has 6 possible sketches whereas the PUFS-SQL-Ordered framework has only 2 sketches to enumerate through. As a consequence the average runtime of the PUFS-X and PUFS-X-Ordered frameworks is significantly higher than the PUFS-SQL-Assited framework, reaching 129.86 seconds and 97.2192 seconds, respectively, in contrast to the 38.16 seconds for the PUFS-SQL-Assited framework. In Table 5.24, we see the total time spent on the same benchmarks, which further demonstrates the difference in performance, with the PUFS-SQL-Ordered framework spending 125.3 seconds in 12 benchmarks in contrast to the 1558.75 seconds spent by the PUFS-X framework and the 1166.63 seconds by the PUFS-X-Ordered framework. We must also note the clear difference between the PUFS-X and PUFS-X-Ordered framework, which demonstrate the improvement of having pruned and ordered the sketches.

All in all, with template benchmarks, the difference between the pruned frameworks is not significant. However, with the free-form benchmarks, we start to see the PUFS-X-Ordered framework spending ten

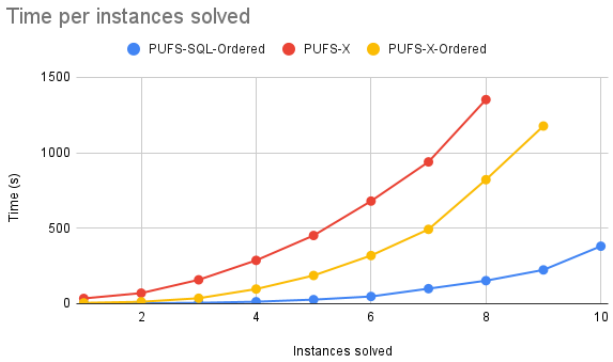


Figure 5.17: PUFS-SQL framework versions: assignment, conditional and data aggregation benchmarks

	PUFS-SQL-Ordered			PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall	Time (s)	Precision	Recall
3-nodes	38.07	1	1	168.9688	1	0.8	130.76	1	0.9

Table 5.25: Evaluation metrics for the PUFS-X frameworks: assignment, conditional and data aggregation benchmarks

times more on the same benchmarks. Recalling that templates should represent a significant percentage of all possible queries, the PUFS-X-Ordered framework continues to be a framework worth pursuing.

Assignment, conditional and data aggregation benchmarks

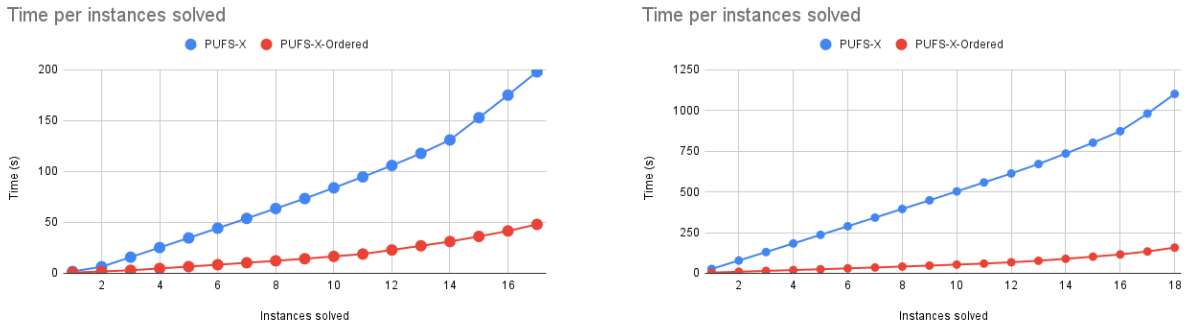
The results for the assignment, conditional and data aggregation benchmarks can be observed in Figure 5.17, Table 5.25 and Table 5.26. The results presented are from the framework PUFS-SQL-Ordered, which has already been presented and analysed, but will be reused here to facilitate the comparison between frameworks. Then, we have the new results obtained by the PUFS-X and PUFS-X-Ordered frameworks.

Note that, similarly to the data aggregation benchmarks, with these benchmarks, the PUFS-X-Ordered framework that is pruned and ordered is not able to reduce the types of nodes accepted, because, from an SQL query, every other type of node may be used. The main difference is the guidance, removing sketches that could never work, such as having a node of type *Assign* after a node of type *DataSet*.

The performance of the frameworks is as expected. The PUFS-X framework is the clear worst, as

	# Benchmarks	PUFS-SQL-Ordered (s)	PUFS-X (s)	PUFS-X-Ordered (s)
3-nodes	8	359.38	1351.75	821.41

Table 5.26: Total runtime of same benchmarks for the PUFS-X frameworks: assignment, conditional and data aggregation benchmarks



(a) Examples with 2-node benchmarks

(b) Examples with 3-node benchmarks

Figure 5.18: PUFS-X framework versions: list manipulation and data aggregation benchmarks

	PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall
2-nodes	11.6435	1	1	2.8265	1	1
3-nodes	61.18	1	1	8.7578	1	1

Table 5.27: Evaluation metrics for the PUFS-X frameworks: list manipulation and data aggregation benchmarks

seen in Figure 5.17, averaging 168.9688 seconds and ending with a recall of 80% not being able to find 2 out of the 10 benchmarks. With the pruning and ordering of sketches, unlike the list manipulation benchmarks, the PUFS-X-Ordered framework despite having better results than the PUFS-X framework, is unable to come close to the performance of the PUFS-SQL-Ordered framework averaging a runtime of 130.76 seconds in contrast to 38.07 seconds. The reason for the big difference in performance is that the ordering and pruning of the sketches when it comes to benchmarks that involve data aggregation benchmarks, can not remove as many sketches from the enumeration process. The recall of the PUFS-X-Ordered framework is of 90% since it was able to find one more solution than the PUFS-X framework.

In Table 5.26, we can observe the total time spent by each framework on the same benchmarks. The PUFS-SQL-Ordered framework spends 359.38 seconds in 8 benchmarks, the PUFS-X framework is the clear worst with 1351.75 seconds and, the PUFS-X-Ordered framework takes 821.41 seconds.

All in all, the PUFS-X-Ordered framework is clearly better than the PUFS-X framework. However, its performance takes a toll for having all features. Considering the increase in the types of programs the PUFS-X-Ordered can solve and that only the data aggregation benchmarks show a significant difference in performance, the framework continues to seem to be worth pursuing.

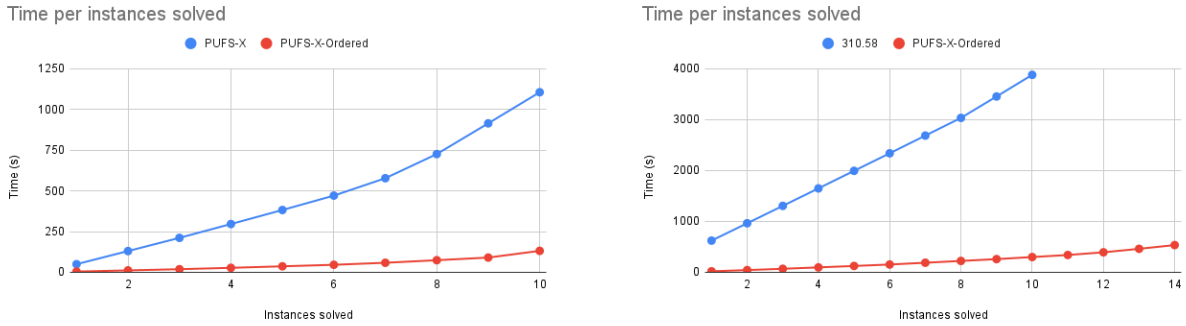
List manipulation and data aggregation benchmarks

The results for the list manipulation and data aggregation benchmarks can be observed in Figure 5.18, Table 5.27 and Table 5.28. The results presented are from the PUFS-X framework versions.

The same pattern can be observe for 2 node and 3 node benchmarks. The PUFS-X framework has a

	# Benchmarks	PUFS-X (s)	PUFS-X-Ordered (s)
2-nodes	17	197.94	48.05
3-nodes	18	1101.24	157.64

Table 5.28: Total runtime of same benchmarks for the PUFS-X frameworks: list manipulation and data aggregation benchmarks



(a) Examples with 2-node benchmarks

(b) Examples with 3-node benchmarks

Figure 5.19: PUFS-X framework versions: assignment, conditional, list manipulation and data aggregation benchmarks

significantly worse performance than the PUFS-X-Ordered framework, specially in 3 node benchmarks, where the PUFS-X framework averaged 61.18 seconds in contrast to the 8.7578 seconds for the PUFS-X-Ordered framework2. For 2 node benchmarks, the average runtime for the PUFS-X framework was 11.6435 seconds in contrast to the 2.8265 seconds for the PUFS-X-Ordered framework. The difference between the pruned and ordered framework and the simple implementation is more drastic with the PUFS-X framework compared to the other frameworks, which is expected due to the exponential nature of the problem and because the framework has 4 different types of nodes. The precision and recall remained at 1 for both frameworks.

All in all, the difference in the results of the frameworks is as expected. We can also observe the expected time the framework requires to solve benchmarks up to 3 node sketches.

Assignment, conditional, list manipulation and data aggregation benchmarks

The results for the benchmarks that encompass all the features can be observed in Figure 5.19, Table 5.29 and Table 5.30. The results presented are from the PUFS-X framework versions.

For 3 node benchmarks, as seen in Table 5.29, the PUFS-X framework averaged 110.553 seconds in solving the 17 benchmarks, whereas the PUFS-X-Ordered framework averaged 13.161 seconds. They both were able to reach all correct solutions to every benchmark resulting in a precision and recall of 1. In Table 5.30, the difference in performance can be consolidated, with the PUFS-X framework spending almost 9 times more time on the same 10 benchmarks spending a total of 1105.53 seconds in contrast to the 131.61 seconds the PUFS-X-Ordered framework needed.

For 4 node benchmarks, the contrast between frameworks increases, which is expected to see in

	PUFS-X			PUFS-X-Ordered		
	Time (s)	Precision	Recall	Time (s)	Precision	Recall
3-nodes	110.553	1	1	13.161	1	1
4-nodes	352.639	0.9091	0.7692	38.2071	1	1

Table 5.29: Evaluation metrics for the PUFS-X frameworks: assignment, conditional, list manipulation and data aggregation benchmarks

	# Benchmarks	PUFS-X (s)	PUFS-X-Ordered (s)
3-nodes	10	1105.53	131.61
4-nodes	10	3535.63	413.08

Table 5.30: Total runtime of same benchmarks for the PUFS-X frameworks: assignment, conditional, list manipulation and data aggregation benchmarks

more complex benchmarks. The PUFS-X framework averaged 352.639 seconds in runtime, was not able to correctly find one solution out of the 14 benchmarks and also not able to find any solution for 3 out of the 14 benchmarks, ending with a precision of 90.91% and a recall of 76.92%. On the other hand, the PUFS-X-ORdered framework averaged 38.2071 seconds and was able to find the correct solution to all benchmarks, ending with a recall and precision of 100%. In Table 5.30, the total time spent on the same benchmarks shows us once again how the PUFS-X framework spent almost 9 times more time than the PUFS-X-Ordered framework on the same benchmarks, spending 3535.63 seconds in contrast to the 413.08 seconds.

All in all, similarly to the list manipulation and data aggregation benchmarks, the difference in the results of the frameworks is as expected. The results of these benchmarks allows us to observe the expected time the framework requires to solve benchmarks that make use of all features up to 4 node sketches.

Chapter 6

Conclusions and Future Work

OutSystems is a software automation platform that allows users to create their applications through graphical interfaces instead of traditional text-based 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 thesis, we proposed a solution through a new pure function synthesizer PUFS+, which supports assignments and conditionals and is based on PUFS. We also proposed PUFS-L and PUFS-SQL, which integrates in PUFS+ the synthesis of list manipulation and data aggregation operators, respectively. Finally, a final version PUFS-X is able to synthesize programs with all mentioned capabilities.

We performed an extensive evaluation of the frameworks, comparing the results between the different variants and using the metrics average runtime, precision and recall. We used a total of the 391 benchmarks ranging in complexity and completeness. We show that PUFS+ improved significantly by being able to solve 90.59% of the benchmarks with a precision of 86.52%, while PUFS was only able to solve 40% with a precision of 81.81%. We also show that, for the frameworks PUFS-L, PUFS-SQL and PUFS-X, the variants which prune and order the sketches significantly improve their performance in all metrics. For instance, for benchmarks ranging all features, when PUFS-X is not pruned nor ordered averages 352.64 seconds in contrast to 38.21 seconds. We also concluded there was a minimal impact on the benchmarks for PUFS-L and PUFS-SQL when adding their respective new features due to the pruning of sketches. However, the PUFS-X framework suffers in performance with benchmarks involving data aggregation queries since there are not as many sketches that can be pruned.

6.1 Future Work

One of the most interesting and enthusiastic parts of this thesis is the multiple ways the work can be improved and extended. For instance, the programs which can be synthesized are still very limited to the number of nodes a sketch has, already showing some difficulty in 3 and 4 nodes. A possible solution is 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. Ambiguity, as expected due to the use of an informal specification, can be a problem. A potential solution is to implement user interaction with the synthesizer to clarify between redundant solutions.

Finally, the synthesizer can be extended to contain more features, such as loops or exception handlers. 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.

Bibliography

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

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

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

[4] K. Ellis and S. Gulwani. Learning to Learn Programs from Examples: Going Beyond Program Structure. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1638–1645. ijcai.org, 2017. URL https://doi.org/10.24963/ijcai.2017/227.

[5] C. C. Green. Application of Theorem Proving to Problem Solving. In D. E. Walker and L. M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 219–240. William Kaufmann, 1969. URL http://ijcai.org/Proceedings/69/Papers/023.pdf.

[6] 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. URL https://doi.org/10.1145/1926385.1926423.

[7] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. 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 50–61. ACM, 2011. URL https://doi.org/10.1145/1993498.1993505.

[8] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, aug 2017. URL https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[9] M. J. H. Heule, O. Kullmann, and A. Biere. Cube-and-Conquer for Satisfiability. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018. URL https://doi.org/10.1007/978-3-319-63516-3.

[10] G. Katz and D. A. Peled. Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms. In S. D. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*, volume 5311 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008. URL https://doi.org/10.1007/978-3-540-88387-6.

[11] T. A. Lau, P. M. Domingos, and D. S. Weld. Version Space Algebra and its Application to Programming by Demonstration. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 527–534. Morgan Kaufmann, 2000. URL https://www.researchgate.net/publication/2237926.

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

[13] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 187–195. JMLR.org, 2013. URL http://proceedings.mlr.press/v28/menon13.html.

[14] T. M. Mitchell. Generalization as Search. *Artif. Intell.*, 18(2):203–226, 1982. URL https://doi.org/10.1016/0004-3702(82)90040-6.

[15] S. Muggleton and L. de Raedt. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, may 1994. URL https://linkinghub.elsevier.com/retrieve/pii/0743106694900353.

[16] 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. URL https://doi.org/10.1007/978-3-030-30048-7.

[17] 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. URL https://doi.org/10.1007/978-3-030-30048-7.

[18] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho. SQUARES: A SQL Synthesizer Using Query Reverse Engineering. *Proc. VLDB Endow.*, 2020. ISSN 2150-8097. URL https://doi.org/10.14778/3415478.3415492.

[19] 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. URL https://doi.org/10.1145/2814270.2814310.

[20] 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. URL https://doi.org/10.1007/978-3-319-21690-4.

[21] A. Solar-Lezama. The Sketching Approach to Program Synthesis. In Z. Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer. URL https://doi.org/10.1007/978-3-642-10672-9.

[22] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008. URL https://dl.acm.org/doi/book/10.5555/1714168.

[23] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326. ACM, 2010. URL https://doi.org/10.1145/1706299.1706337.

[24] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. 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 492–503. ACM, 2011. URL https://doi.org/10.1145/1993498.1993557.

[25] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013. URL https://doi.org/10.1145/2509578.2509586.

[26] P. van der Tak, M. Heule, and A. Biere. Concurrent Cube-and-Conquer - (Poster Presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer, 2012. URL https://doi.org/10.1007/978-3-642-31612-8.

[27] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24,*

2009, Vancouver, Canada, Proceedings, pages 364–374. IEEE, 2009. URL https://doi.org/10.1109/ICSE.2009.5070536.

Appendix A

Tables of DSL operations

Function Signature	Description	Examples
add(x: Numeric, y: Numeric): Numeric	Returns the addition of the Numeric x and Numeric y.	add(20,10) = 30 add(0,5) = 5
sub(x: Numeric, y: Numeric): Numeric	Returns the subtraction of x by y.	sub(20,10) = 10 sub(0,5) = -5
mul(x: Numeric, y: Numeric): Numeric	Returns the multiplication of x by y.	mul(20, 10) = 200 mul(0, 100) = 0
div(x: Numeric, y: Numeric): Numeric	Returns the division of x by y.	div(20, 10) = 2 div(10, 1) = 10
abs(x: Numeric): Numeric	Returns the absolute value of x.	abs(20) = 20 abs(-15) = 15
sqrt(x: Numeric): Numeric	Returns the square root of the number x.	sqrt(4) = 2 sqrt(49) = 7
trunc(x: Numeric): Numeric	Returns the truncation of the number x.	trunc(1.0218) = 1 trunc(0.9999) = 0
round(x: Numeric): Numeric	Returns the round of the number x.	round(1.0218) = 1 round(0.9999) = 1
round2(x: Numeric, y: Numeric): Numeric	Returns the round of the number x by y decimals.	round2(1.0218, 2) = 1.02 round2(0.4599) = 0.46
mod(x: Numeric, y: Numeric): Numeric	Returns the modulus of the number x by y.	mod(5, 2) = 1 mod(16, 4) = 0
power(x: Numeric, y: Numeric): Numeric	Returns the power of x over y.	power(3, 2) = 9 power(2, 3) = 8
max(x: Numeric, y: Numeric): Numeric	Returns the maximum value between x and y.	max(-5, 5) = -5 max(1, 20) = 20
min(x: Numeric, y: Numeric): Numeric	Returns the minimum value between x and y.	min(1, 1) = 1 min(-20, 2) = -20
sign(x: Numeric): Numeric	If $x < 0$ then return -1, else if $x > 0$ return 1, else return 0.	sign(-10) = -1 sign(99) = 1

Table A.1: Numeric DSL operations of PUFS+

Function Signature	Description	Examples
add(*x*: Text, *y*: Numeric): Text add(*x*: Text, *y*: Boolean): Text add(*x*: Numeric, *y*: Text): Text add(*x*: Boolean, *y*: Text): Text	Returns the addition of *x* and *y*	add(”a ” , 1) = ”a 1” add(”y”, False) = ”yFalse” add(1, ” a”) = ”1 a” add(False, ”y”) = ”Falsey”
substr(*x*: Text, *y*: Numeric, *z*: Numeric): Text	Retrieves first position of *y* at or after *z* characters in *x*. Returns -1 if there are no occurrences of *y* in *x*.	substr(”Test”, 1, 2) = ”es” substr(”123”, 0, 1) = ”12”
replace(*x*: Text, *y*: Text, *z*: Text): Text	Returns Text *x* after replacing all Text occurences of *y* with *z*	replace(”aba”,”a”, ”c”) = ”cbc” replace(”x”,”x”, ”123”) = ”123”
concat(*x*: Boolean, *y*: Text): Text	Returns the concatenation of *x* with *y*.	concat(”a”,”b”) = ”ab” concat(”12”,”4”) = ”124”
chr(*x*: Numeric): Text	Returns a single-character string corresponding to the *x* character code.	chr(88) = ”X”
length(*x*: Text): Numeric	Returns the length of the Text *x*.	add(True,”x”) = ”Truex” add(False,”y”) = ”Falsey”
toLower(*x*: Text): Text	Returns the Text *x* in lowercase.	toLower(”AbCd”) = ”abcd” toLower(”AAA”) = ”aaa”
toUpper(*x*: Text): Text	Returns the Text *x* in uppercase.	toUpper(”AbCd”) = ”ABCD” toUpper(”aaa”) = ”AAA”
trim(*x*: Text): Text	Removes all the leading and trailing space characters (’ ’) from the Text *x*.	trim(” a ”) = ”a” add(” 1 2 ”) = ”1 2”
trimStart(*x*: Text): Text	Removes all the leading space characters (’ ’) from the Text *x*.	trimStart(” a”) = ”a” trimStart(” a ”) = ”a ”
trimEnd(*x*: Text): Text	Removes all the trailing space characters (’ ’) from the Text *x*.	trimEnd(”a ”) = ”a” trimEnd(” a ”) = ” a”

Table A.2: Text DSL operations of PUFS+

Function Signature	Description	Examples
gt/gte(*x*: Numeric, *y*: Numeric): Boolean gt/gte(*x*: Text, *y*: Numeric): Boolean gt/gte(*x*: Text, *y*: Boolean): Boolean gt/gte(*x*: Numeric, *y*: Text): Boolean gt/gte(*x*: Boolean, *y*: Text): Boolean gt/gte(*x*: Text, *y*: Text): Boolean	True if *x* is *greater*/*greater or equal* than *y*. False otherwise.	gt(10, 1) = True gte(”aa”, 1) = True gt(”aa”, True) = False gte(1, ”aa”) = False gt(True, ”aa”) = True gte(”aaa”, ”aa”) = True
eq/diff(*x*: Numeric, *y*: Numeric): Boolean eq/diff(*x*: Text, *y*: Numeric): Boolean eq/diff(*x*: Text, *y*: Boolean): Boolean eq/diff(*x*: Text, *y*: Text): Boolean	True if *x* is *equal*/*different* than *y*. False otherwise.	eq(1, 1) = True diff(”aa”, 2) = False eq(”aaaa”, True) = True diff(”aaa”, ”aa”) = True
and(*x*: Boolean, *y*: Boolean): Boolean	Returns True if both *x* and *y* are True.	and(True, True) = True and(True, False) = False
or(*x*: Boolean, *y*: Boolean): Boolean	Returns True if either *x* or *y* are True.	or(True, False) = True or(False, False) = False
not(*x*: Boolean): Boolean	Returns True if *x* is False. Otherwise returns False.	not(True) = False not(False) = True

Table A.3: Boolean DSL operations of PUFS+

Function Signature	Description	Examples
ListAppend(*x*: List, *y*: Numeric): List ListAppend(*x*: List, *y*: Text): List ListAppend(*x*: List, *y*: Boolean): List ListAppend(*x*: List, *y*: Structure): List	Returns the List *x* with *y* appended	ListAppend([], 2) = [2]
ListInsert(*x*: List, *y*: Numeric, *z*: Numeric): List	Returns the List *x* with *y* inserted in the *z* index.	ListInsert([1], 2, 0) = [2, 1]
ListInsert(*x*: List, *y*: Text, *z*: Numeric): List		ListInsert([”a”], ”b”, 1) = [”a”, ”b”]
ListInsert(*x*: List, *y*: Boolean, *z*: Numeric): List		ListInsert([id: 1], id: 2, 1) = [id: 1, id: 2]
ListAppendAll(*x*: List, *y*: List): List	Returns a List of *x* with *y* appended.	ListAppendAll([1], [2, 2]) = [1, 2, 2]
ListSort(*x*: List, *y*: Boolean): List	Returns the List *x* ordered in ascending order if *y* is True. Otherwise in descending order.	ListSort([1, 3, 2], True) = [1, 2, 3] ListSort([1, 3, 2], False) = [3, 2, 1]
ListRemove(*x*: List, *y*: Numeric): List	Returns the List *x* with the index *y* removed.	ListRemove([1, 3], 0) = [3] ListRemove([1, 3, 2], 1) = [1, 2]
ListFilter(*x*: List, *y*: CmpLambda): List	Returns the List *x* filtered by the OpLambda *y*.	ListFilter([1, 3, 2], x: x $>$ 1) = [3, 2]
ListIndexOf(*x*: List, *y*: CmpLambda): Numeric	Returns the index of the first occurrence of the OpLambda *y* in the List *x*.	ListIndexOf([1, 3, 2], x: x==2) = 2 ListIndexOf([1, 3, 2], x: x¿1) = 1
ListAll(*x*: List, *y*: CmpLambda): Boolean	Returns True if all elements of the List *x* satisfy the OpLambda *y*. Otherwise it returns False.	ListAll([1, 3, 2], x: x¿0) = True ListAll([1, 3, 2], x: x¿1) = False
ListAny(*x*: List, *y*: CmpLambda): List	Returns True if any of the elements in the List *x* satisfy the OpLambda *y*. Otherwise it returns False.	ListAny([1, 3, 2], x: x¿1) = True ListAny([1, 3, 2], x: x¡0) = False
ListClear(*x*: List): List	Returns an empty List.	ListClear([1, 3, 2]) = []
ListDistinct(*x*: List): List	Returns the List *x* without any duplicate elements.	ListDistinct([1, 2, 2]) = [1, 2] ListDistinct([1]) = [1]
ListDuplicate(*x*: List): List	Returns the List *x* with each element duplicated.	ListDuplicate([1]) = [1, 1] ListDuplicate([1, 3]) = [1, 1, 3, 3]

Table A.4: Outsystems built-in DSL operations of PUFS-L

Function Signature	Description	Examples
ListMap(*x*: List, *y*: OpLambda): List	Returns the List *x* with the OpLambda *y* applied to each element.	ListMap([1], x: add(x, 1)) = [2] ListSort([1, 3], x: mul(x, 3)) = [3, 9]
getElementList(*x*: List, *y*: Numeric): BasicType	Returns the element of index *y* of the List *x*.	getElementList([”hi”], 0) = ”hi” getElementList([1, 3], 1) = 3

Table A.5: Custom DSL operations of PUFS-L

Function Signature	Description
Select(*x*: Table): List	Returns a list of rows of the table *x*.
SelectOrderBy(*x*: Table, *y*: Col, , *z*: ConstBool): List	Returns a list of rows of the table *x* ordered by the column *y* in ascending order if *z* is True, otherwise in descending order.
natural_join(*x*: Table, *y*: Table): Table	Returns a table with the natural join of *x* and *y*.
natural_join3(*x*: Table, *y*: Table, *z*: Table): Table	Returns a table with the natural join of *x*, *y* and .
natural_join4(*x*: Table, *y*: Table, *z*: Table, *w*: Table): Table	Returns a table with the natural join of *x*, *y*, *z* and *w*.
inner_join(*x*: Table, *y*: Table, *z*: JoinCondition): Table	Returns a table with the inner join of *x* and *y* and with the condition *z*.
anti_join(*x*: Table, *y*: Table, *z*: Cols): Table	Returns a table with the anti join of *x* and *y* and with the columns *z*.
left_join(*x*: Table, *y*: Table): Table	Returns a table with the left join of *x* and *y*.
union(*x*: Table, *y*: Table): Table	Returns a table with the union of *x* and *y*.
intersect(*x*: Table, *y*: Table, *z*: Col): Table	Returns a table with the intersection of *x* and *y* on column *z*.
semi_join(*x*: Table, *y*: Table): Table	Returns a table with the semi join of *x* and *y*.
summarise(*x*: Table, *y*: SummariseCondition, *z*: Cols): Table	Returns a table with summarisation condition *y* of table *x*, grouped by columns *z*.
mutate(*x*: Table, *y*: SummariseCondition): Table	Returns the table *x* mutated by the summarise condition *y*.

Table A.6: PUFS-SQL freeform operations

Function Signature	Description
Select(*x*: Table): List	Returns a list of rows of the table *x*.
SelectCondition(*x*: Table, *y*: FilterCondition): List	Returns a list of rows of the table *x* filtered by y.
SelectConditionOrderBy(*x*: Table, *y*: FilterCondition, *z*: Col, *w*: ConstBool): List	Returns a list of rows of the table *x* with the filter *y* ordered by the column *z* in ascending order if *w* is True, otherwise in descending order.
SelectLeftJoinCondition(*x*: Table, *y*: Table, *z*: FilterCondition): List	Returns a list of rows of the left join of table *x* and *y*, with the filter *z*.
SelectLeftJoinConditionOrderBy(*x*: Table, *y*: Table, *z*: FilterCondition, *w*: Col, *v*: ConstBool): List	Returns a list of rows of the left join of table *x* and *y*, with the filter *z*, ordered by the column *w* in ascending order if *v* is True, otherwise in descending order.
SelectInnerJoinCondition(*x*: Table, *y*: Table, *z*: JoinCondition, *w*: FilterCondition): List	Returns a list of rows of the inner join of table *x* and *y* on the condition *z*, with the filter *w*.
SelectInnerJoinConditionOrderBy(*x*: Table, *y*: Table, *z*: JoinCondition, *w*: FilterCondition, *v*: Col, *s*: ConstBool): List	Returns a list of rows of the inner join of table *x* and *y* on the condition *z*, with the filter *w*, ordered by the column *v* in ascending order if *s* is True, otherwise in descending order.
SelectCrossJoinCondition(*x*: Table, *y*: Table, *z*: CrossJoinCondition, *w*: FilterCondition): List	Returns a list of rows of the cross join of table *x* and *y* on the condition *z*, with the filter *w*.
SelectCrossJoinConditionOrderBy(*x*: Table, *y*: Table, *z*: CrossJoinCondition, *w*: FilterCondition, *v*: Col, *s*: ConstBool): List	Returns a list of rows of the cross join of table *x* and *y* on the condition *z*, with the filter *w*, ordered by the column *v* in ascending order if *s* is True, otherwise in descending order.

Table A.7: PUFS-SQL template operations

Enum Signature	Description
Col	List of columns of the tables provided by the user. For example, ”id” or ”name”.
Op	Operations | and & for the filter conditions of a query.
FilterCondition	Filter conditions for a given table. For example, ”id == 1” or ”id $>$ 5”.
JoinCondition	Join conditions for a join operation between tables. For example, ”name1 == name2”.
CrossJoinCondition	Cross join conditions for a cross join operation between tables. For example, ”name == name.other”.
SummariseCondition	Summarise conditions for a summarise operation in a table. For example, ”sum(price)”.

Table A.8: Added DSL enums to PUFS-SQL