TÉCNICO LISBOA

Pure Function Synthesis in the OutSystems Platform

Catarina Pina de Almeida Coelho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Maria Inês Camarate de Campos Lynce de Faria
Doctor Miguel Ângelo da Terra Neves

Examination Committee

Chairperson: Professor Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Professor Maria Inês Camarate de Campos Lynce de Faria
Member of the Committee: Professor João Fernando Ferreira

January, 2021

Acknowledgments

First of all, I would like to thank my supervisors, Professor Inês Lynce and Doctor Miguel Neves, for all of their support during this very challenging phase, for always trying to help me in every way they could, and for their valuable advice, without which I am certain I would not be able to close this chapter of my academic life.

I would like to thank Miguel Ventura for welcoming me at OutSystems and for sharing the most insightful drawings of graphs on a wall, and for all of the advice and insights that helped me improve the quality of this work.

I am very grateful to Margarida and Ricardo for being by my side while facing this challenge, sharing all the difficulties and achievements, and most of all, for all the times when everything felt harder and they were always there with a helping hand. I would like to thank to some of my oldest friends, Alexandra, Ricardo and Joana, for sharing my concerns about program synthesis even though they had no clue what I was talking about!

I would like to thank my parents and my sister, for always supporting and encouraging me, and always having a kind word when they were needed the most. You inspire me everyday to be better and do better.

Finally, I would like to thank António for always being by my side.

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

Resumo

A síntese de programas consiste em gerar automaticamente um programa a partir de uma especificação usada para definir a intenção do utilizador. A plataforma OutSystems é uma plataforma de desenvolvimento *low-code* que permite o desenvolvimento de aplicações através de uma interface gráfica.

A plataforma OutSystems permite que a lógica das aplicações seja implementada através de fluxos de ações, que por sua vez podem ser usados para executar várias operações complexas e recorrentes, tais como as operações de manipulação de dados. Para tal, as funções puras podem ser usadas em expressões usadas pela linguagem OutSystems de forma a executar estas operações. As funções puras são um tipo de função que não tem quaisquer efeitos secundários e o seu valor de retorno é determinado pelos seus valores de *input*. Contudo, escrever este tipo de funções pode tornar-se uma tarefa repetitiva e aborrecida devido à sua frequência, e pode ainda ser uma tarefa difícil para utilizadores com menos experiência.

Neste documento apresentamos o PUFS, um sintetizador de funções puras, que dado um conjunto de exemplos input-output, como especificação do comportamento da função desejada, sintetiza uma função pura. A nossa solução consiste numa combinação entre *sketches* de programas como uma representação parcial de uma função parcial e procura enumerativa em conjunto com Satisfazibilidade Módulo Teorias (SMT), para preencher os *sketches* de programas de forma a obter a função completa. Avaliámos a nossa solução em instâncias do mundo real, mostrando resultados experimentais promissores para muitas das funções puras recorrentes e comuns.

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

Abstract

Program synthesis consists in automatically generating a program from a specification used to define user intent. The OutSystems platform is a low-code development platform which allows the development of applications through a graphical user interface.

The OutSystems platform allows business logic to be implemented through action flows, which can be used to perform several complex and recurrent operations, such as data wrangling operations. In order to do this, pure functions can be used within OutSystems language expressions to perform these operations. Pure functions are a type of functions that have no side-effects and their returned value is determined by its inputs. However, writing this type of functions might become a tedious and repetitive task due to its recurrence, and might even be a difficult task for less experienced users.

In this work we present PUFS, a pure function synthesizer that given a set of input-output examples, as a specification of the function's desired behavior, synthesizes a pure function. Our solution consists of a combination between program sketches as a representation of a partial function and enumeration based search alongside Satisfiability Modulo Theories (SMT) to fill the sketches in order to obtain the complete function. The proposed solution was evaluated on a set of real-world examples, showing promising results for recurrent and common pure functions.

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

Contents

Acknowledgments iii

Resumo v

Abstract vii

List of Figures xi

List of Tables xiii

1 Introduction **1**

1.1 Motivating Example 2

1.2 Contributions 3

1.3 Organization 4

2 Fundamental Concepts **5**

2.1 Program Synthesis 5

2.1.1 User Intent 5

2.1.2 Program Space 6

2.1.3 Search Techniques 6

2.2 Satisfiability Modulo Theories 8

3 Related Work **10**

3.1 Inductive Synthesis 10

3.1.1 Programming-by-Example 10

3.1.2 Ambiguity Resolution 11

3.2 Program Sketches 12

3.2.1 Sketch-based Program Synthesis 12

3.2.2 Counterexample Guided Inductive Synthesis 13

3.3 Enumeration-Based Program Synthesis 13

3.3.1 Line-based Encoding 14

3.3.2 Encoding Variables 15

3.3.3 Constraints 15

4 Pure Function Synthesis **18**

4.1 Problem Formulation 18

4.2 Sketch Generation 22
4.3 Sketch Completion 26
4.3.1 Input-Output Examples and Domain Specific Language (DSL) 27
4.3.2 K-Trees Enumeration 28
4.3.3 Line-based Encoding with Conditionals 29
4.3.4 Decider 32

5 Evaluation **33**
5.1 Examples Generation 33
5.1.1 Examples Pre-selection 33
5.1.2 Examples Categorization 35
5.1.3 Generation of input-output examples 37
5.2 Experimental Results 39
5.2.1 Evaluation 39
5.2.2 Discussion 40

6 Conclusions and Future Work **42**

Bibliography **48**

List of Figures

1.1 Motivating example of a pure function in the OutSystems platform 3

2.1 Program synthesis dimensions 6
2.2 Enumerative search program synthesis 7

3.1 Sketch-based program synthesis 13
3.2 Counterexample guided inductive synthesis 13
3.3 Formula $Price + (Price * TaxRate)$ from motivating example in Section 1.1 as a k-tree. . 14
3.4 K-tree from Figure 3.3 represented as lines using the line-based representation. 14

4.1 Example of flow in Service Studio component of OutSystems platform 21
4.2 Pure Function Synthesizer Framework 22
4.3 Partial flow example 22
4.4 Enumeration of sketches given the desired size of the flow 23
4.5 Free nodes possibilities. 23
4.6 Sketch enumeration phase when there are more than three nodes left to reach the size limit of the sketch. 24
4.7 Sketch enumeration phase when there are three nodes left to reach the bound 25
4.8 Sketch enumeration phase when there are two nodes left to reach the bound 25
4.9 Sketch enumeration phase when there is one node left to reach the bound 26
4.10 Sketch enumeration phase when the bound is reached 26
4.11 Overview of the PUFS synthesis process. 27
4.12 An example AST 28
4.13 Example AST of figure 4.12 as a k-tree. 28
4.14 Enumeration of k-trees for a given sketch with two holes to fill. 29

5.1 Flow with loop 34
5.2 Venn Diagram for Pre-selection of Examples 35
5.3 Number of instances for each size category 36
5.4 Example flows interpreter. 38
5.5 Number of instances solved throughout time for a timeout of 120 seconds. 41

List of Tables

1.1 Input-output examples for motivating example 2

4.1 Description of the built-in functions used for synthesis 18
4.2 Possible input-output examples for example flow in figure 4.1 21

5.1 Total for each pre-selection category and overlapping of each category 34
5.2 Number of instances for each category of flows with conditional nodes and without conditional nodes 37
5.3 Number of instances that are supported by the synthesizer for each category of flows with conditional nodes and without conditional nodes 37
5.4 Total number of instances that are supported by the synthesizer without repeated instances 38
5.5 Comparison of number of instances solved for the different timeout values. 40

Chapter 1

Introduction

OutSystems platform is a low-code development platform which allows the development of applications through a graphical user interface. Its main goal is to provide an easier and faster experience in development and integration of web and mobile applications. The OutSystems platform allows the implementation of business logic using actions which can be used later in other action flows. An action flow is a set of operations represented by nodes, such as access to a database, assignment of variables, among others, that implements the logic of the application. Unlike regular action flows, action flows can be used within OutSystems language expressions, making them a special case of these type of flows, thus very useful to perform complex data transformations that are recurrent throughout the application.

Pure functions are a type of functions that have no side-effects, where the return value is only determined by its input values, as in functions in traditional programming languages. Such functions can be math functions, such as the cosine function, that given the same input always return the same output value.

Although the platform provides an easier experience that abstracts the user from the code writing task, it also relies on the use of action flows to prevent the user from having to repeat the same operations. Since the implementation of pure functions in these flows is a frequent element in every application, it makes sense to develop an automation of this process.

Code generation has been one of the main recurrent research fields throughout the years, its relevance has become higher and led to the appearance of new research fields and techniques, one of them being program synthesis. Program synthesis consists in automatically generating a program that satisfies a specification provided by the user to express its intent, i.e., the desired behavior of the program.

It becomes clear that this technique can be quite useful in the context of our problem, since we want to be able to facilitate the generation of these functions in the platform by some sort of automation, given that these are recurrent tasks throughout the platform. However, providing such specification in program synthesis may be a difficult task, as sometimes users know what a task should do, but do not know how to express it a functional language. Given that, two main approaches for providing a specification on a program's desired behaviour have stood out: specifications as input-output examples and natural language descriptions. In the first approach the user provides the desired output values for each input value, and for the second approach the user provides a description of the behavior of the program.

One of the main properties of pure functions is that their output is conditioned by the input. Hence, we can see that the inputs have a great influence in the behavior of these functions. As such, we have chosen an approach based on input-output examples as specification.

Having a specification in the form of input-output examples enables users with no specific knowledge about a problem, to provide a specification that is accessible according to their knowledge base. As such, they are easy to create and can be used in a wide range of problems. Furthermore, besides being easy to create they also represent an specification that has a simpler implementation when it comes to the synthesis process implementation, as in comparison to natural language based specifications for instance. Although those might represent an easier specification to provide in some cases, it would require a more complex solution. However the use of examples as a specification also comes with a disadvantage: ambiguity of the solution, i.e., the synthesizer might be able to find a program that satisfies the examples but the program does not correspond to the desired behaviour.

Pure functions in the action flows are not code fragments, and so they do not have the structure of typical functions as in other programming languages. These functions are represented as a flow of several nodes in which each node has an operation to be executed within the function. Therefore, its representation might not be so straight forward for those who are more familiarized with a writing approach. Due to the regularity of performing these tasks, as well as the characteristics that these type of functions have, it makes sense to automatize this process using synthesis techniques.

1.1 Motivating Example

Suppose the user wants to synthesize a pure function by providing a set of input-output examples. In particular, the user wants to synthesize a function that allows to calculate the price of an article with the corresponding tax, given the price and the tax rate. If the price is less then zero then the price should be returned as zero. If the price is greater than zero and as well as the tax rate the returned price should be the price including the tax. If the price is greater than zero but the tax rate is not, then the price with no tax applied.

However, for the user, this might not be a solution easy to come by, since the user might not have a programming background or enough experience with the platform to know how to translate the desired function in the form of a flow. To overcome this challenge, the user provides the specification using input-output examples. A possible set of input-output examples for this specific case would be:

Table 1.1: Input-output examples for motivating example

Input	Output
19.90, 0.23	24.477
-1.0, 0.23	0.00
-1.0, -0.23	0.00
1.50, -0.23	1.50
5.60, 0.06	5,936

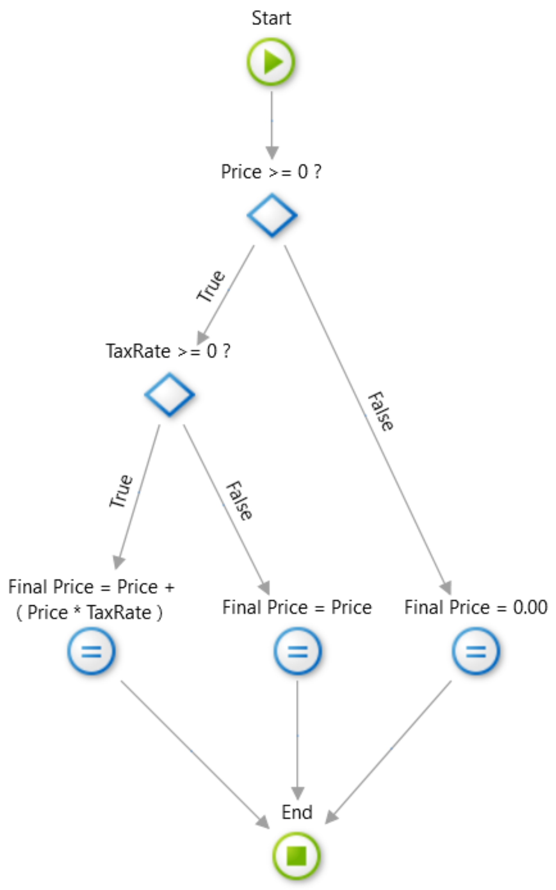

Figure 1.1: Motivating example of a pure function in the OutSystems platform

According to the provided specification, the synthesizer will try to find a program which given every input will return the corresponding output. Figure 1.1 shows a possible solution in the platform for the given specification.

1.2 Contributions

In this work, we survey the state of the art in program synthesis and implement a Programming-by-Example-based Pure Function Synthesizer, PUFS, developed on the top of a state-of-the-art synthesis framework Trinity [17] incorporated with the line representation present by Orvalho et. al as well as the use of sketches as the underlying program representation [21]. The synthesizer employs a mixture of Programming-by-Example with enumerative search. PUFS' main goal is to synthesize pure functions in the OutSystems platform from input-output examples. We gathered benchmarks from the real-world and evaluated the performance of our synthesizer.

This thesis makes the following contributions:

- We propose a PBE-based program synthesizer whose goal is to solve the problem of synthesizing pure functions in the OutSystems platform.
- We implement a tool named PUFS, which employs the use of sketches as a partial representation

of the programs and enumerative search to complete the partial programs in order to obtain the complete programs.

1.3 Organization

This document is organized as follows. Chapter 2 presents the fundamental concepts and notations used throughout the document. Chapter 3 provides some insights on related work such as Inductive Synthesis (Section 3.1), Program Sketches (Section 3.2) and Enumeration-Based Program Synthesis (Section 3.3).

Afterwards, Chapter 4 presents the proposed solution, the Pure Function Synthesizer PUFS, namely its architecture and main components. Chapter 5 provides a description of the benchmarks used as well as the evaluation methods and results at Section 5.2.

Finally, Chapter 6 presents the conclusions about the developed work and some possible future work.

Chapter 2

Fundamental Concepts

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

2.1 Program Synthesis

Given a specification used to express the user intent, program synthesis is the task of automatically generating a program that satisfies that specification. Different types of specification include input-output examples [1, 2, 6, 9, 20, 24, 31], logical formulas [14, 16] and natural language [2, 5, 31].

In program synthesis there are three main dimensions, as illustrated in Figure 2.1: expressing user intent, program space and search techniques, which are described in more detail in sections 2.1.1, 2.1.2 and 2.1.3, respectively.

2.1.1 User Intent

As mentioned previously, to perform program synthesis there must be a way for the user to express his intent. The user intent indicates the desired behavior of the program to be generated by the synthesizer.

Specification Given an input x and an output value y, a specification ϕ is the description of the user's intent, such that $\phi(x, y)$ is True if and only if y is the desired output value for x.

Despite all the progresses in program synthesis solutions, expressing user intent still remains a significant challenge. The first approaches on program synthesis, such as deductive synthesis, required the user intent to be expressed using a complete formal specification, which in most cases is harder than writing the program itself.

Using a complete specification might become as challenging as the underlying programming task. However, when not specific enough, there might be more than one program that satisfies the provided

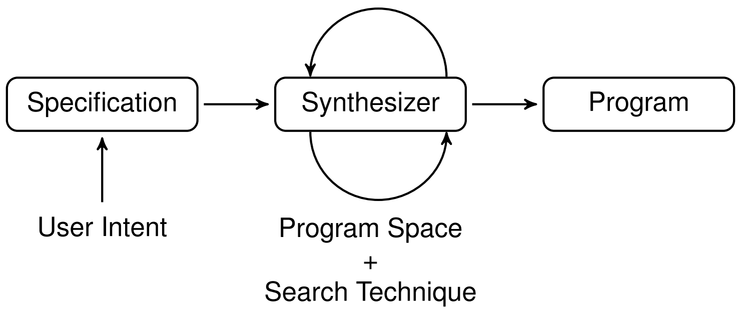

Figure 2.1: Program synthesis dimensions

specification and end up with a program that is not the desired one, due to the ambiguity of the specification.

The goal is to find an approach that allows finding the desired solution without the need for a very complex specification, i.e., find a balance between the completeness and ease of formulation of the specification.

2.1.2 Program Space

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

Program Space is the space containing the set of all programs that can be written using a given programming language.

The program space for a given programming language is infinite, which leads to another challenge: the dimension of the program space. In order to tackle this challenge, one of the many possible approaches is to restrict the program space by imposing an upper bound on the number of lines or instructions that a program can have. However, the size of this restricted program space grows exponentially as the upper bound grows or as more components are added to the language.

A possible approach to reduce the restricted program space is making use of a pruning technique, such as domain-specific heuristics, restricting the program space using some program complexity metrics such as size or restrict the programs language using a Domain-Specific Language (DSL).

Domain-Specific Language defines both the syntax and the semantics of the language in which the synthesized programs are written, providing the appropriate notions and abstractions for a particular domain or problem.

2.1.3 Search Techniques

In order to find the intended program, one needs to search the program space for a program that satisfies the specification. The specification and the knowledge about the context of the problem are used in order

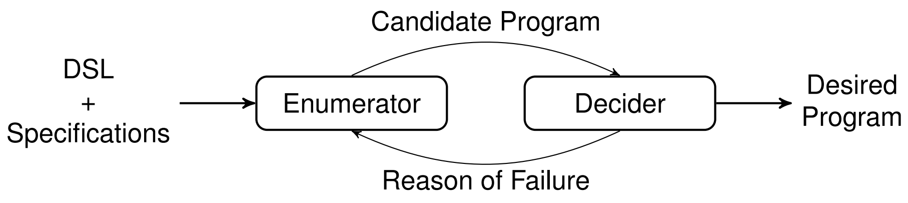

Figure 2.2: Enumerative search program synthesis

to guide the search process. To do so, there are four main search techniques in program synthesis, which are described in more detail in this section.

Enumerative Given a specification and a Domain Specific Language (DSL), the enumerative based approach consists in enumerating the programs that are in the search space using some heuristic to define the order in which they are enumerated, which can be program size, complexity, among others. Then, for each program, it checks if it satisfies or not the specification. In Figure 2.2 we can see an illustration of the enumeration process.

The enumerator is responsible for enumerating the candidate programs. These candidate programs are sent to a verifier which checks the consistency of the program according to the specification provided by the user. If the program is consistent, then it is returned to the user; otherwise, the enumerator must provide a new program to be verified.

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

Deductive The deductive approach follows a top-down search and the *divide-and-conquer* technique. The divide-and-conquer technique consists of recursively reducing the synthesis problem into simpler sub-problems and combining the results of solving the sub-problems.

Given a synthesis problem of synthesizing an expression e of the form $F(e_1, e_2)$ that must satisfy a specification, it reduces the problem to simpler sub-problems e_1 and e_2 and finds a solution for each one separately. The solution for both e_1 and e_2 is then combined to derive the desired expression e.

Constraint Solving This technique consists of two steps: constraint generation and constraint resolution, as in [14]. Constraint generation consists of building a logic formula whose solution is a program that satisfies the specification. According to Gulwani et.al [15], there are 3 main kinds of methods for constraint generation: invariant-based, path-based, and input-based.

Constraint resolving consists of solving the logical constraints generated during the constraint generation phase, which usually is achieved by the use of a Boolean Satisfiability (SAT) or a Satisfiability Modulo Theory (SMT) solver. SMT is explained in more detail in section 2.2.

Statistical Statistical techniques make use of machine learning, genetic programming, Markov Chain Monte Carlo (MCMC) sampling or probabilistic inference. *Machine Learning* is mainly used to support other search techniques such as enumerative search and deduction, by learning an heuristic to guide the search. On the other hand, given an initial program, *MCMC* is used to search for the desired program, by making some local changes which allow to obtain a program that is a optimized version of the initial candidate.

Genetic programming, as the name indicates, is based on the biological evolution process where programs are treated as individuals in a population and evolved through the use of genetic operators. This evolution is achieved by mutation and/or crossover of a said population of individual programs. Mutation introduces random changes in the program, while crossover shares pieces of code among the programs in the population.

2.2 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 $f^{\mathcal{M}} : M^n \to M$, 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 $p^{\mathcal{M}} : M^n \to \{true, false\}$.

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

Chapter 3

Related Work

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

3.1 Inductive Synthesis

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

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

3.1.1 Programming-by-Example

Programming-by-Example (PBE) is a sub-field of program synthesis that focuses on input-output example based specifications.

One of PBE's main goals is automating certain classes of programming tasks, which has proven extremely useful for end-users since it is easier to provide examples rather than a formal specification of the constraints, but also very useful to developers since it provides a tool for automating repetitive and tedious programming tasks in the form of informal specifications.

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

Example 2. The following set of input-output examples specifies the behavior of a program that receives as input a list of integers, and returns a list with the even values in the list.

Input	Output
[1, 2, 5, 8]	[2, 8]
[1, 3, 11]	[ ]
[2, 10, 22]	[2, 10, 22]

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

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

3.1.2 Ambiguity Resolution

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

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

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

3.2 Program Sketches

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

3.2.1 Sketch-based Program Synthesis

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

A *sketch* expresses the high-level structure of an implementation but has holes which represent the low-level details. The key idea is to create an abstraction from the source code that clearly defines the semantics but not the syntax, this is, the *sketch* abstracts out names and operations from a program, but keeps the program's structure, the order in which it executes methods, types of arguments and its return values. This approach is know as *programming with sketches* [28].

Example 3. Program illustrating basic structure of a sketch.

```
void main(int x){
    int y = x * ??;
    assert y == x + x;
}
```

The above example demonstrates the basic structure of a sketch, composed by three elements: a main procedure *main*, holes *??* and assertions *assert y == x + x*. The assertions encode the specification of the intended behavior. These express the properties which the synthesized program should satisfy for all possible inputs. The holes are then replaced with values that ensure the correctness of the generated program.

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

Also, it allows the user to focus on the algorithmic properties of the implementation rather than the low-level details. Solar-Lezama et al. [28] show that this approach improves the productivity and performance of programming tasks.

The sketch-based synthesis process can be split in two stages: sketch generation and sketch completion, as illustrated in Figure 3.1. The first process consists in generating a sketch, using an automated sketch generation technique [8], in which the synthesizer enumerates the sketches according to some complexity metric, such as the sketch size, and a given DSL. Followed by the filling of the holes, with the use of a synthesizer to fill each hole with an according expression in order to generate a complete program, which corresponds to the sketch completion stage. This process is repeated until a valid solution is found according to the given specification.

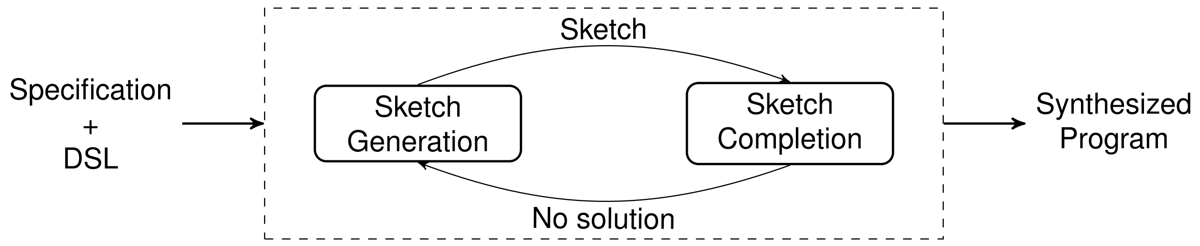

Figure 3.1: Sketch-based program synthesis

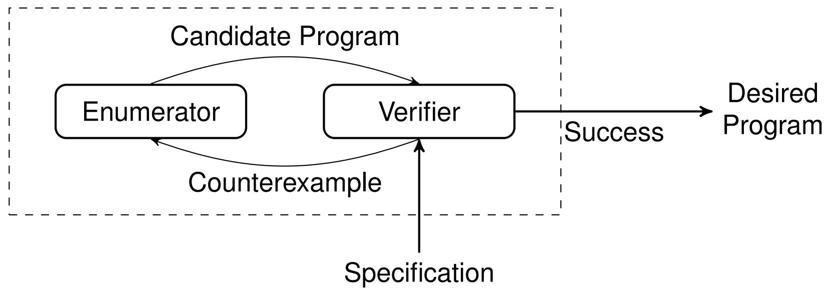

Figure 3.2: Counterexample guided inductive synthesis

3.2.2 Counterexample Guided Inductive Synthesis

Along with the *skecth* Solar-Lezama proposed a new synthesizer, which uses a technique known as *Counterexample Guided Inductive Synthesis* (CEGIS) [27].

As shown in Figure 3.2 the enumerator starts by providing a candidate program which is then verified against the specification. If the verification succeeds, i.e. the candidate program satisfies the input-output examples, the program is considered correct, and is returned to the user. If the verification fails, the verifier checks if there exists an input for which the specification is not satisfied; if it exists, the verifier sends the corresponding input-output pair as a counterexample to the enumerator. After some iterations, the synthesizer will have enough counterexamples in order to return a valid candidate which will be accepted and returned by the verifier to the user.

3.3 Enumeration-Based Program Synthesis

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

As shown in Figure 2.2 the enumeration-based technique has two main components: an enumerator and a decider. The enumerator enumerates all the possible programs for a DSL given as input. For each enumerated program the decider will evaluate if it satisfies the specification provided by the user. For

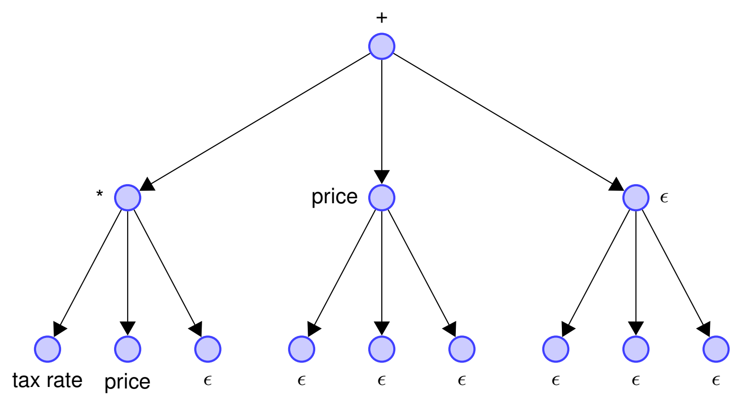

Figure 3.3: Formula $Price + (Price * TaxRate)$ from motivating example in Section 1.1 as a k-tree.

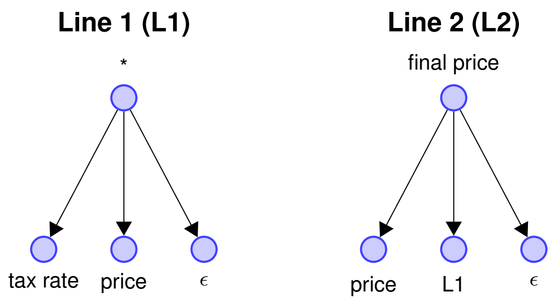

Figure 3.4: K-tree from Figure 3.3 represented as lines using the line-based representation.

the particular case of PBE, this evaluation performed by the decider is done by executing the enumerated program using the input examples and checking if the output matches the corresponding output examples. If the output does not match the expected one we consider that program to be infeasible.

3.3.1 Line-based Encoding

This section describes the line-based encoding proposed by Orvalho et. al. [21], based on the tree-based encoding presented by Martins et. al. [17] used in the Trinity synthesis framework.

Given a DSL, Trinity enumerates candidate programs in order to find a program that is consistent with the input-output examples provided by the user. To perform this search, it uses a structure capable of representing all the candidate programs for a given DSL. For that purpose, each program is represented using a tree structure referred to as k-tree of depth n, where each node has exactly k children, k being the greatest arity among the DSL constructs. Figure 3.3 illustrates the tree representation of the formula used to calculate the price with the tax rate in the pure function shown in the motivating example presented in section 1.1 as a k-tree, given a DSL where the greatest arity among all operators is 3.

The enumeration is performed in order of increasing complexity, i.e., the enumerator enumerates the trees in an increasing depth order as trees of smaller depths are exhausted. So the more complex the program to synthesize the bigger the depth of the tree representing that program.

The encoding proposed by Orvalho et. al. [21] represents a program as a sequence of lines where each line represents an operation from the DSL. In this case, each line is represented using a k-tree of depth one, rather than having one single k-tree representing the complete program. Using this encoding the tree representation used in figure 3.3 would turn into the representation illustrated in figure 3.4. So the tree that was previously a 3-tree of depth 2 is now represented using two 3-trees of depth 1. This representation allows for more complex programs to be represented using less nodes given to the fact that each k-tree can be used as an argument of other trees. Therefore the line-based approach scales better than the tree-based representation in Trinity.

Although there are other possible tree-representations such as binary trees or rose trees, the binary tree representation has exactly 2 children so it limits the greatest arity of the operators in the DSL to be 2. As for rose-trees, which have an unbounded number of children per node, these do not have an encoding in SMT and do not provide an advantage in comparison to the use of k-trees for a known DSL. For those reasons we focused in applying k-tree based encodings for the purpose of this dissertation.

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 a symbol of the DSL to each node. The SMT encoding of the trees follows.

3.3.2 Encoding Variables

Let D be the DSL, $Prod(D)$ the set of production rules in D and $Term(D)$ the set of terminal symbols in D. Furthermore, $Types(D)$ represents the set of types used in D and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ corresponds to the return type of production rule s.

Consider a program with n lines, 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 line i;
- $T = \{t_i : 1 \leq i \leq n\}$: each variable t_i represents the return type of line 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 in line i;

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

3.3.3 Constraints

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 line in the program. Let $Ret = \{ret_i : 1 \leq i \leq n\}$ represent the set of return symbols in the program, then $\Sigma = Prod(D) \cup Term(D) \cup Ret$.

Furthermore, each symbol is assigned a unique positive identifier. Let $id : \Sigma \rightarrow \mathbb{N}_0$ be a one-to-one mapping function that maps each symbol in Σ to a unique positive identifier and $tid : Types(D) \rightarrow \mathbb{N}_0$ be a one-to-one mapping function that maps each symbol type to a unique positive identifier. Finally, since some operations in the DSL have a smaller arity than k, the empty symbol ϵ is introduced, so that every leaf node has an assigned symbol. We assume $id(\epsilon) = 0$.

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

$$\forall\, 1 \leq i \leq n : \bigvee_{p \in Prod(D)} op_i = id(p) \tag{3.1}$$

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

$$\forall\, 1 \leq i \leq n,\ p \in Prod(D) : (op_i = id(p)) \implies (t_i = tid(Type(p))) \tag{3.2}$$

Arguments. The arguments of an operation i must be either terminal symbols or return symbols from previous lines.

$$\forall\, 1 \leq i \leq n,\ 1 \leq j \leq k : \bigvee_{s \in Term(D) \,\cup\, \{ret_r : r < i\} \,\cup\, \epsilon} a_{ij} = id(s) \tag{3.3}$$

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

$$\begin{gathered}\forall\, 1 \leq i \leq n,\ p \in Prod(D),\ 1 \leq j \leq arity(p),\ 1 \leq r < i : \\ ((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \implies (t_r = tid(Type(p, j)))\end{gathered} \tag{3.4}$$

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

$$\begin{gathered}\forall\, 1 \leq i \leq n,\ p \in Prod(D),\ 1 \leq j \leq arity(p), \\ s \in \{r \in Term(D) : Type(r) \neq Type(p, j)\} : \\ (op_i = id(p)) \implies \neg(a_{ij} = id(s))\end{gathered} \tag{3.5}$$

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

$$\begin{gathered}\forall\, 1 \leq i \leq n,\ p \in Prod(D),\ arity(p) < j \leq k : \\ (op_i = id(p)) \implies (a_{ij} = id(\epsilon))\end{gathered} \tag{3.6}$$

Output. Let $Type(out)$ denote the type of the program's output and $P_{out} \subseteq Prod(D)$ be the subset of productions rules which return type matches $Type(out)$, i.e., $P_{out} = \{p \in Prod(D) : Type(p) = Type(out)\}$. Since the last line ($n^{th}$ line) corresponds to the program's output, the operation of the last line must be one of the productions in P_{out}.

$$\bigvee_{p \in P_{out}} (op_n = id(p)) \tag{3.7}$$

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

$$\forall s \in I : \bigvee_{1 \leq i \leq n} \bigvee_{1 \leq j \leq k} (a_{ij} = id(s)) \tag{3.8}$$

Chapter 4

Pure Function Synthesis

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

4.1 Problem Formulation

This thesis was developed within the context of the OutSystems[1] platform. OutSystems is a low-code development platform which provides a graphical user interface for the development of mobile and web applications, while allowing easy integration with other existing systems and the use of traditional textual programming (e.g. JavaScript, SQL) when needed. Its main goal is to enable an easier and faster development experience of enterprise-level applications.

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

An OutSystems expression is composed by operands and operators. The operands can be a literal (e.g. strings, numbers, boolean values, etc.), any element available in the scope of the current expression, such as local variables, or function calls, or sub-expressions. The operators can be of type numeric, logic and Boolean, among others. However, in this work we are focusing on synthesizing pure functions that use built-in types[2] such as Integer, Decimal, Text and Boolean only, although there are others, and built-in functions[3] such as Math, Numeric and Text only, described in table 4.1.

Table 4.1: Description of the built-in functions used for synthesis

[1]`https://www.outsystems.com`

[2]`https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Data_Types/Available_Data_Types`

[3]`https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic/Built-in_Functions`

Built-In Function	Description	Example
+(n:Value, m:Value)	Returns the sum between n and m.	2 + 1 = 3 1.3 + 0.5 = 1.8 ”ab” + ”c” = ”abc”
-(n:Number, m:Number)	Returns n minus m.	2 - 1 = 1 1.3 - 0.5 = 0.8
*(n:Number, m:Number)	Returns n times m.	2 * 1 = 2 1.3 * 0.5 = 0.65
/(n:Number, m:Number)	Returns n divided by m.	2 / 1 = 2.0 1.3 / 0.5 = 2.6
>(n:Value, m:Value)	Returns the result of the logical operation > between n and m.	2 > 1 = True -0.1 > 0.5 = False ”ab” > ”c” = True
<(n:Value, m:Value)	Returns the result of the logical operation < between n and m.	2 < 1 = False -0.1 < 0.5 = True ”ab” < ”c” = False
>=(n:Value, m:Value)	Returns the result of the logical operation >= between n and m.	2 >= 1 = True 0.3 >= 0.5 = False ”ab” >= ”cd” = True
<=(n:Value, m:Value)	Returns the result of the logical operation <= between n and m.	2 <= 1 = False 0.3 <= 0.5 = True ”ab” <= ”cd” = True
=(n:Value, m:Value)	Returns the result of the logical operation = between n and m.	2 = 1 = False 0.3 = 0.3 = True ”ab” = ”ab” = True
<>(n:Value, m:Value)	Returns the result of the logical operation <> between n and m.	2 <> 1 = True 0.3 <> 0.3 = False ”ab” <> ”ab” = False
and(n:Boolean, m:Boolean)	Returns the result of the logical operation and between n and m.	True and False = False 5 > 2 and 1 < 2 = True
or(n:Boolean, m:Boolean)	Returns the result of the logical operation or between n and m.	True or False = True 5 > 2 or 1 < 2 = True
Abs(n:Decimal)	Returns the absolute value of n.	Abs(-10.2) = 10.2
Mod(n:Decimal, m:Decimal)	Returns the remainder of n divided by m.	Mod(10, 3) = 1 Mod(4, 3.5) = 0.5
Power(n:Decimal, m:Decimal) the power of m.	Returns n raised to Power(100, 2) = 10000 Power(-10.89, 2.3) = 0	
Round(n:Decimal, m:Integer)	Returns the Decimal number n rounded to a specific number of fractional digits.	Round(-10.89) = -11 Round(-5.5) = -6 Round(9.3) = 9 Round(9.123456789, 4) = 9.1235
Sqrt(n:Decimal)	Returns the square root of n.	Sqrt(2.3) = 1.51657508881031
Trunc(n:Decimal)	Returns the Decimal number n truncated to integer by removing the decimal part of n.	Trunc(-10.89) = -10 Trunc(7.51) = 7

Max(n:Decimal, m:Decimal)	Returns the largest number between n and m.	Max(-10.89, -2.3) = -2.3 Max(10.89, 2.3) = 10.89
Min(n:Decimal, m:Decimal)	Returns the smallest number between n and m.	Min(-10.89, -2.3) = -10.89 Min(10.89, 2.3) = 2.3
Sign(n:Decimal)	Returns -1 if n is negative; 1 if ’n’ is positive; 0 if ’n’ is 0.	Sign(-10.89) = -1 Sign(2.3) = 1 Sign(0.0) = 0
Chr(c:Integer)	Returns a single-character string corresponding to the c character code.	Chr(88) = ”X”
Concat(t1:Text, t2:text)	Returns the concatenation of two Texts: t1 and t2	Concat(”First string”, ”last string”) = ”First stringlast string” Concat(””, ””) = ””
Length(t:Text)	Returns the number of characters in Text t.	Length(”First string”) = 12 Length(””) = 0
Replace(t1:Text, t2:text, t3:Text)	Returns Text t1 after replacing all Text occurrences of t2 with t3.	Replace(”Hello”, ”xx”, ””) = ”Hello” Replace(”Hello world”, ”Hello”, ”Bye”) = ”Bye world” Replace(”Hello world”, ”Hello”, ””) = ” world”
Substr(t:Text, pos:Integer, len:Integer)	Returns a sub-string of t beginning at pos zero-based position and with len characters.	Substr(”First string”, 2, 4) = ”rst ” Substr(”First string”, 0, 100) = ”First string” Substr(”First string”, 11, 3) = ”g” Substr(”First string”, Length(”First string”), 0) = ”” Substr(”First string”, 2, 0) = ””
ToLower(t:Text)	Converts Text t to the equivalent lowercase text.	ToLower(”Hello”) = ”hello”
ToUpper(t:Text)	Converts Text t to the equivalent uppercase text.	ToUpper(”Hello”) = ”HELLO”
Trim(t:Text)	Removes all leading and trailing space characters (’ ’) from Text t.	Trim(” Hello ”) = ”Hello” Trim(”Hello ”) = ”Hello”
TrimEnd(t:Text)	Removes all trailing space characters (’ ’) from Text t.	TrimEnd(” Hello ”) = ” Hello” TrimEnd(”Hello ”) = ”Hello”
TrimStart(t:Text)	Removes all leading space characters (’ ’) from Text t.	TrimStart(” Hello ”) = ”Hello ” TrimStart(”Hello ”) = ”Hello ”

Although there are more built-in types and built-in functions supported by the platform, we start by trying to synthesize functions that contain the ones mentioned before so that we start by evaluating our synthesizer on a smaller set of options and consequently on simpler programs. So that afterwards, using the results of the evaluation of our synthesizer using this set of types and functions we are able to better decide on how to improve our solution and if it is feasible to include more types and functions without compromising performance. This selection was made based on this and also on the fact that a lot of the instances retrieved from the real-world frequently used these functions and types.

We are mainly focused in synthesizing pure functions, but for the scope of this thesis we are focused on the ones with just conditional expressions in form of If statements and assignment expressions without loops. The assignment expressions assign a value to a given variable. On the other hand, If

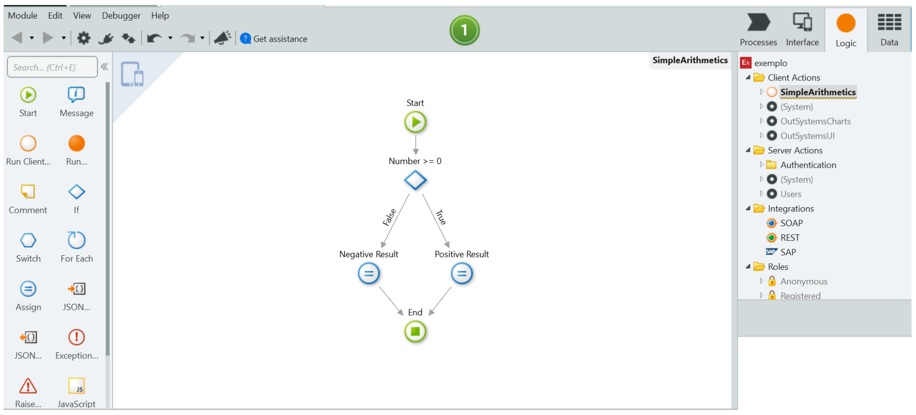

Figure 4.1: Example of flow in Service Studio component of OutSystems platform

statements consist of an expression to be evaluated in order to condition the control-flow of the function. Synthesis of functions containing loops was not considered due to the fact that this will be a novel approach in synthesizing pure functions in the OutSystems platform, i.e., there isn't a solution for this problem that allows to do so in the present, and as such we will start by trying to synthesize simpler functions to verify if is possible to improve to more complex functions.

In Figure 4.1 we illustrate an example of a flow that represents a pure function with an "if" node, which evaluates an expression to check if an input value is greater than or equal to zero, followed by two assignment nodes. If the input is greater than or equal to zero the assignment node Positive Result will assign the value 1 to the output value, otherwise the assignment node Negative Result will assign the value 0 to the output value.

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

Let (x_i, y_i) be input-output pairs in a set of N input-output examples $\mathcal{X}$. The goal is to synthesize f such that:

$$\bigwedge_{(x_i,y_i)\in\mathcal{X}} f(x_i) = y_i \tag{4.1}$$

Example 4. Assuming we wish to synthesize the flow in figure 4.1 with a Start node, an If node , two Assign nodes and an End node, where the goal is, given an integer as input, to return 1 if the integer is greater than or equal to zero and 0 otherwise. A possible set of input-output pairs is:

Table 4.2: Possible input-output examples for example flow in figure 4.1

Input	Output
100	1
-10	0
0	1
-57	0

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

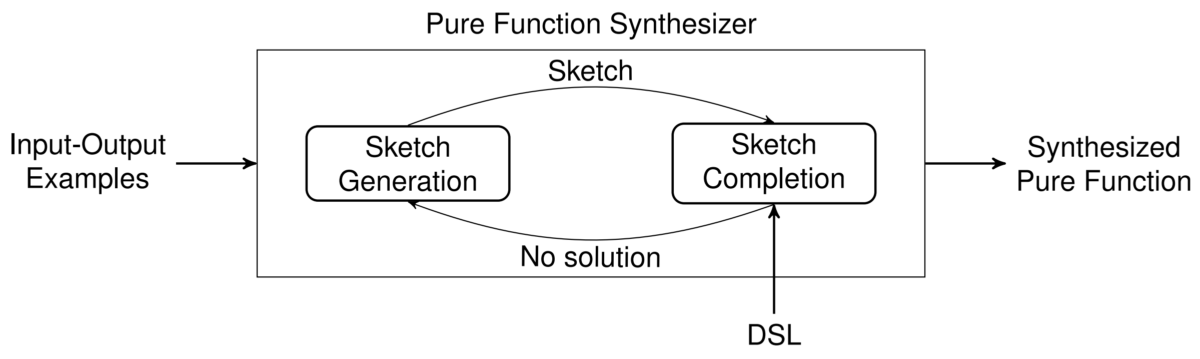

Figure 4.2: Pure Function Synthesizer Framework

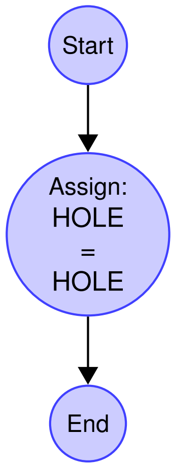

Figure 4.3: Partial flow example

The structure of the input file and assemble of the DSL is further described in subsection 4.3.1.

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

The overview of the architecture of our framework is illustrated in figure 4.2.

4.2 Sketch Generation

The sketch generation consists in generating a sketch of a flow, i.e., a flow with holes in place of the assignment and If node expressions as illustrated in figure 4.3. This generation process consists of a enumerative approach that enumerates several sketches up to a pre-specified size. The size of a flow corresponds to the number of nodes in that flow.

In our approach, graphs are used as a representation of the flows, since a flow is akin to the control-

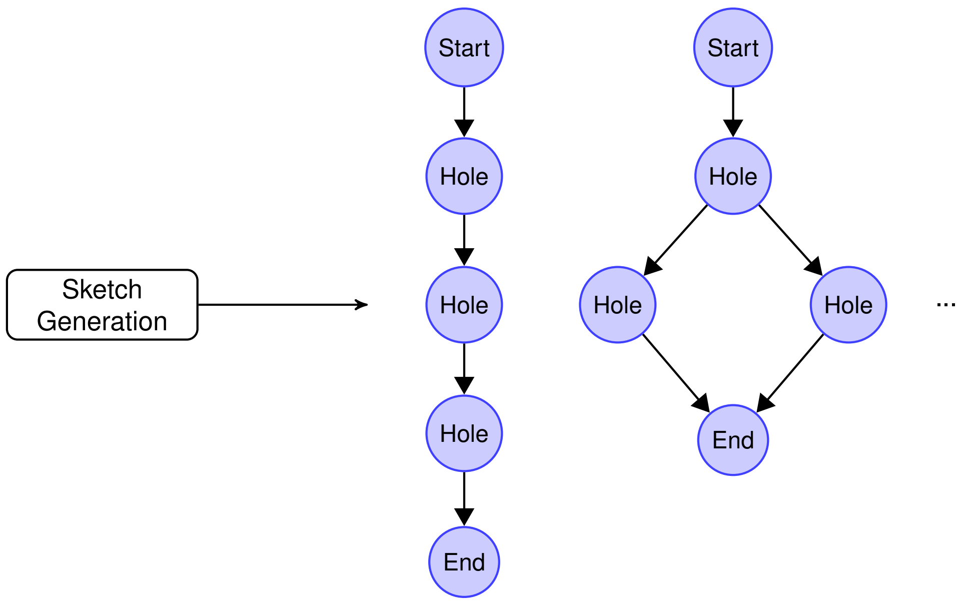

Figure 4.4: Enumeration of sketches given the desired size of the flow

flow graph of an application. Given that, we consider a flow to be a graph. Also, the graphs allow us to have a representation that gathers all the necessary information to enumerate the sketches, such as the neighborhood of each node, which expression is associated to the node, among others. Therefore, in this section, when we refer to a flow, we refer to its structure as a graph. We consider the size of a flow to be the number of nodes of the corresponding graph. Figure 4.4 shows the sketches that would be generated for a fixed size of 5.

Given the desired size of the flow, the enumerator follows a recursive approach conditioned by the current size of the flow and the desired final size. We consider 5 possibilities, that correspond to when there are 0, 1, 2, 3 or more nodes away from reaching the desired flow size.

At each step of the recursion we check if there are If and non-If free nodes. Free nodes correspond to nodes which have no outgoing edges. We consider non-If free nodes to be nodes of type Start or Assign that have no outgoing edges and If free nodes to be If nodes with at most one outgoing edge or none, since If nodes must have 2 outgoing edges corresponding to the ”True” and ”False” edges. Both possibilities are shown in figure 4.5.

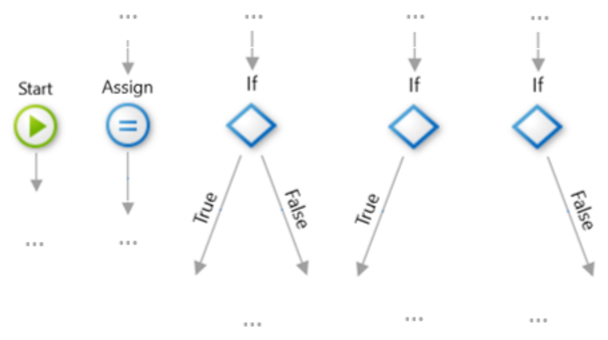

Figure 4.5: Free nodes possibilities.

In order to distinguish between the edges that connect Assign nodes and If nodes, the edges have an attribute 'kind' which has 3 possible values: 'Connector' representing the outgoing edges of non-If nodes, and the values 'True' and 'False' to represent the outgoing edges of If nodes.

At each new recursive step we check how many nodes are missing in the current graph in order to achieve the desired size. Then, based on the size of the current graph and the corresponding free nodes we add the possible nodes among Assign, If and End nodes. At the end of each step a new recursion begins using the updated graphs.

More than 3 nodes. If more than 3 nodes are missing, the enumerator generates the possible flows that result from adding 1 Assign node, or 1 If node to any non-If node or If node with no outgoing edges, as illustrated in figure 4.6. Despite not being represented in the figure, the non-If nodes include the Start Node. The same applies for the False branch scenarios for the If nodes. This applies to the following examples in this section.

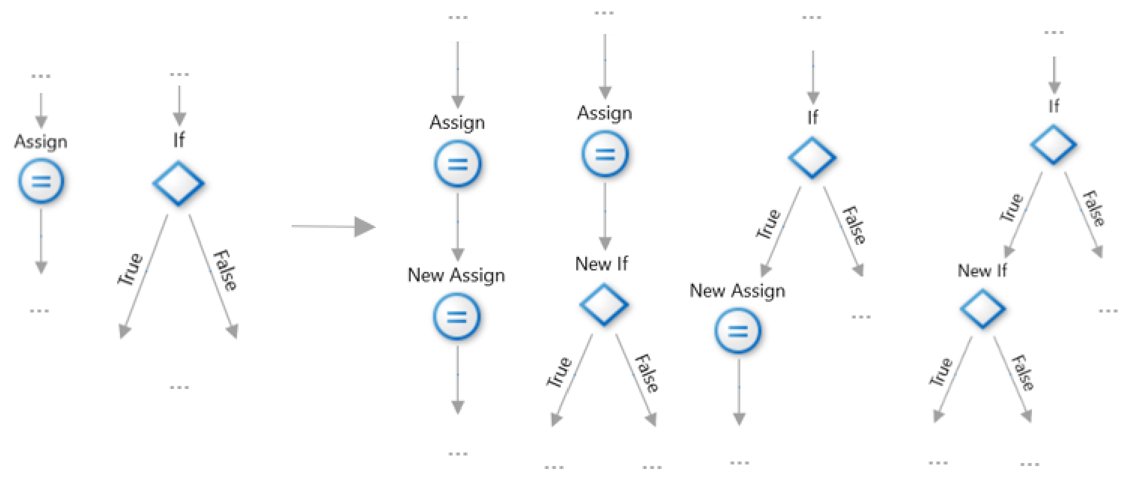

Figure 4.6: Sketch enumeration phase when there are more than three nodes left to reach the size limit of the sketch.

3 Nodes. When there are 3 nodes missing, it generates the possible flows by adding 3 Assign nodes or adding 1 If node with branches to Assign nodes. For If free nodes it also generates the flow with 2 or 3 more assign nodes connected to the free If node, or adds an If node with branches to Assign nodes. All these possibilities are illustrated in figure 4.7. The possibility of adding a If node with branches to two other If nodes is not considered in this case. Since there are only 3 nodes missing to reach the size of the flow, by adding 3 If nodes, the only remaining possibility would be to connect all the branches from those nodes to an End node. Therefore, in practice, adding those nodes would not represent any additional functionality to the function itself.

2 Nodes. When 2 nodes are missing there are two possibilities, either add 2 Assign nodes or add 1 If and 1 Assign node. Both options are illustrated in figure 4.8. The possibility of adding 2 If nodes is not considered in this case. Since there are only 2 nodes missing to reach the size of the flow, by adding 2 If nodes, the only remaining possibility would be to connect all the branches from those nodes to an

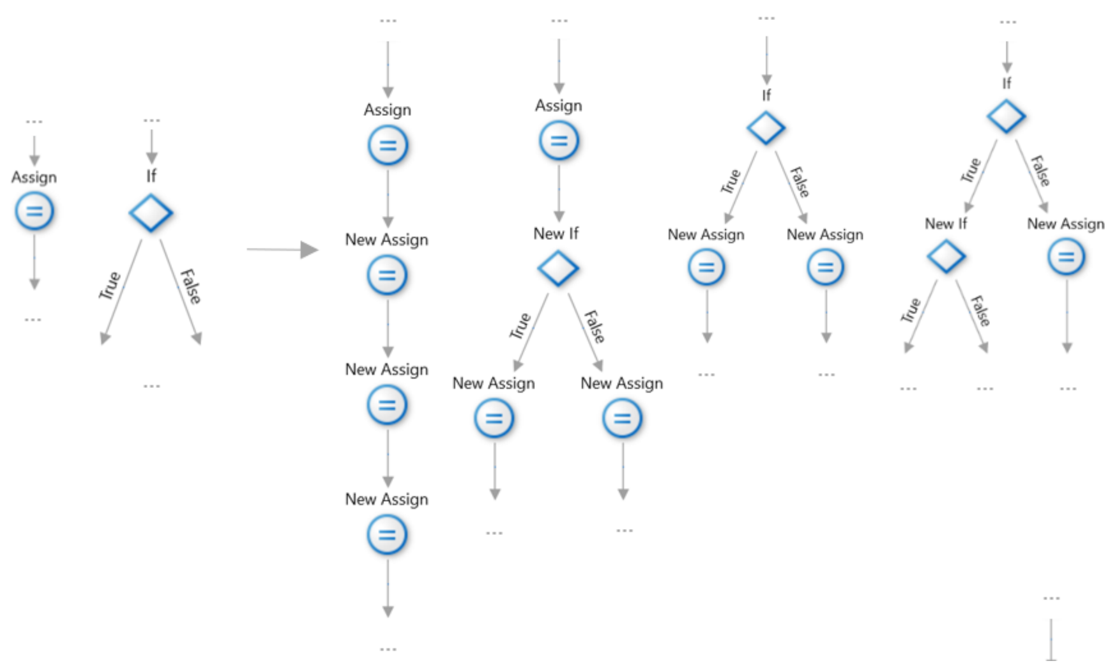

Figure 4.7: Sketch enumeration phase when there are three nodes left to reach the bound

End node. Therefore, in practice, adding those nodes would not represent any additional functionality to the function itself. The possibility of adding 1 If Node and 1 Assign node to and If free node is not considered as well, for the same reasons mentioned before.

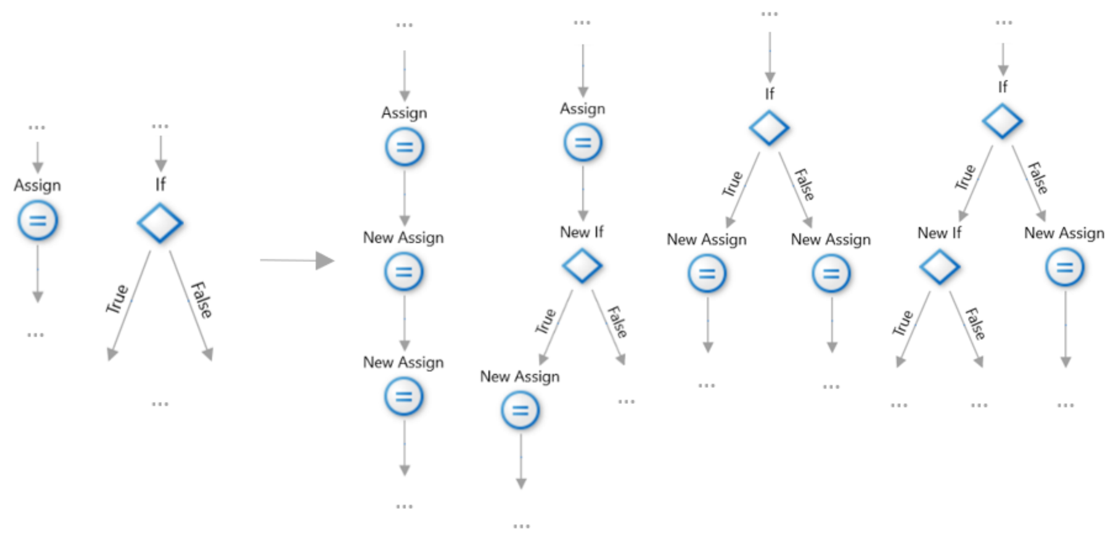

Figure 4.8: Sketch enumeration phase when there are two nodes left to reach the bound

1 Node. When 1 node is missing, the enumerator prioritizes the extensions of conditional nodes. If it finds If free nodes, the next step is to check if there are any If nodes with no outgoing edges in order to create an edge between the existing If and an existing Assign node. Otherwise, it just adds a new Assign node to the free If node. When there are non-If free nodes we simply create a branch from the free node to a new Assign node. All the mentioned combinations are illustrated in figure 4.9.

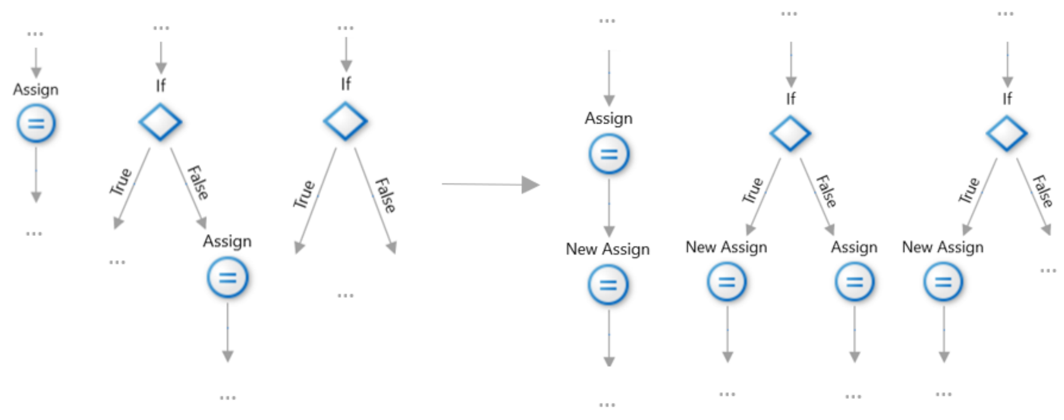

Figure 4.9: Sketch enumeration phase when there is one node left to reach the bound

0 Nodes. When there are no nodes missing, i.e., the size of the current graph is the same as the size limit for the sketch, the sketch enumerator searches for all the free nodes and creates an edge from each of those free nodes to an End node, as illustrated in figure 4.10.

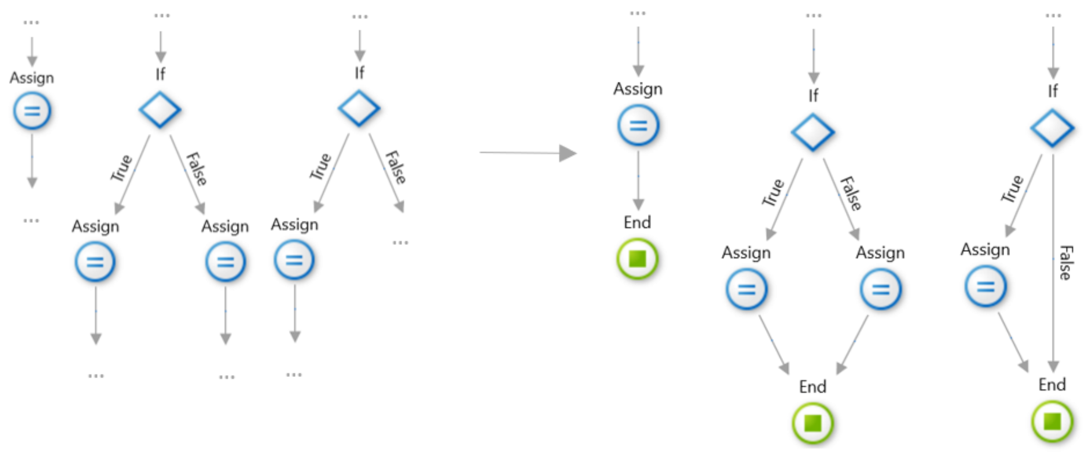

Figure 4.10: Sketch enumeration phase when the bound is reached

However, as the size of the flow increases so does the number of possible sketches generated, which can lead to a lower performance of the generator since it has to enumerate more possibilities. In order to overcome this challenge, a possible solution would be to apply symmetry breaking techniques in order to reduce the number of sketches that have to be generated.

4.3 Sketch Completion

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

During the sketch completion stage we enumerate several candidate programs, in the form of k-trees

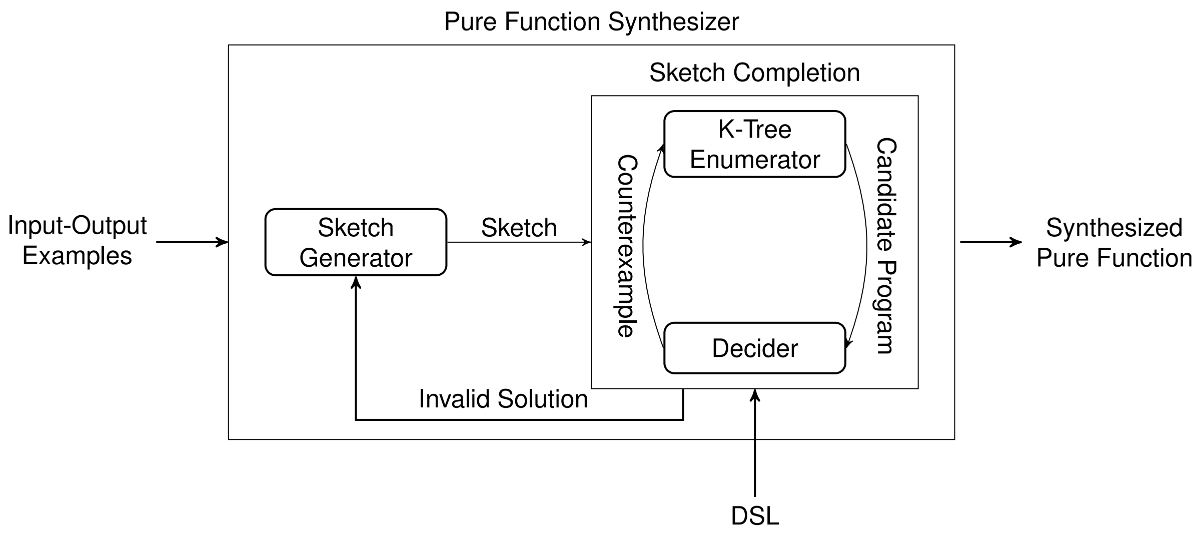

Figure 4.11: Overview of the PUFS synthesis process.

(section 4.3.2), from a given DSL, further described in section 4.3.1. Afterwards, we fill the sketch holes and verify if the complete flow is consistent with all the input-output examples. If the flow is consistent with the examples, we consider that a solution was found, otherwise, the sketch enumerator provides another sketch and the process is repeated until a solution is found. This process is illustrated in figure 4.11

4.3.1 Input-Output Examples and Domain Specific Language (DSL)

In order to synthesize the expressions used to fill in the sketch holes, we need to define a DSL that includes the definition of operators and values that can be used to build the desired program. These operators and operands will form the expressions used to complete the sketch to obtain a complete and valid flow. The set of operators and operands defined in the DSL contain the built-in functions and built-in types mentioned before, along with arithmetic and logical comparison operators, as shown in Table 4.1. However, there are components of the DSL that are defined based on the input file given by the user. This file includes the following: the input-output examples, the number of inputs and outputs of the desired program and constants.

Number of inputs and outputs of the desired program. The first line specifies the number of inputs and outputs of the desired program, so that the DSL can be built according to that information.

Input-Output Examples. The second line contains the input-output examples to be used as specification for the synthesizer.

Constants. In the third and final line of the file, the user can provide constants to be used in the program, in order to find a correct solution that contains those same constants, otherwise the synthesizer does not search for those programs.

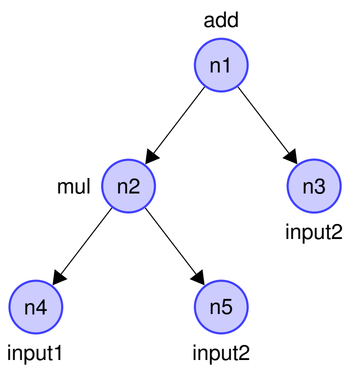

Figure 4.12: An example AST

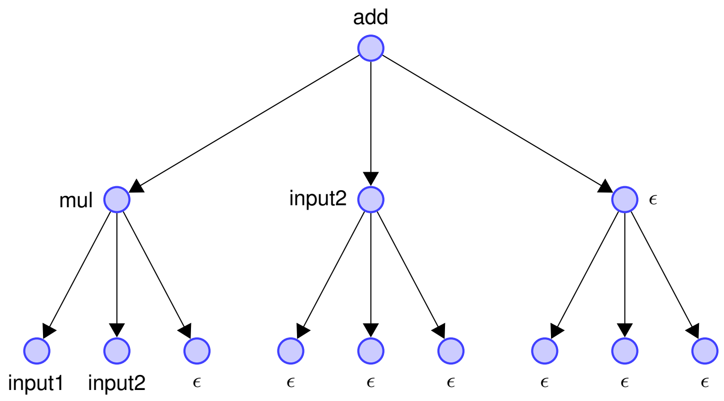

Figure 4.13: Example AST of figure 4.12 as a k-tree.

4.3.2 K-Trees Enumeration

Once the input file, with the input-output examples, along with the DSL are provided, the enumeration of the candidate programs takes place. The enumeration of the candidate programs is guided by a sketch, i.e., the sketch is the flow's graph with holes instead of expressions.

As mentioned before in section 3.3, in order to perform the enumeration of candidate programs, we need to use a structure that is capable of representing every possible program in the DSL. Programs are often represented using their Abstract Syntax Tree (AST) representation.

An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of a program, where each internal node represents an operator, and the children represent the respective operands. For instance, the AST shown in Figure 4.12 corresponds to the program $add(mul(input1, input2), input2)$.

As mentioned previously in section 3.3 k-trees are a tree representation used in enumeration-based program synthesis due to its ability of representing every possible program for a given DSL. Therefore, k-trees are the representation used in PUFS. A k-tree is a tree of depth d, where every internal node has exactly k children and every leaf node is at depth d.

For the current DSL supported by PUFS, the maximum arity among all DSL constructs is 3, meaning every k-tree will have 3 children, as shown in Figure 4.13.

In order to enumerate the possible programs using k-trees, the synthesizer encodes the trees as an SMT formula. A complete program can be extracted from a model of the SMT formula. A model that

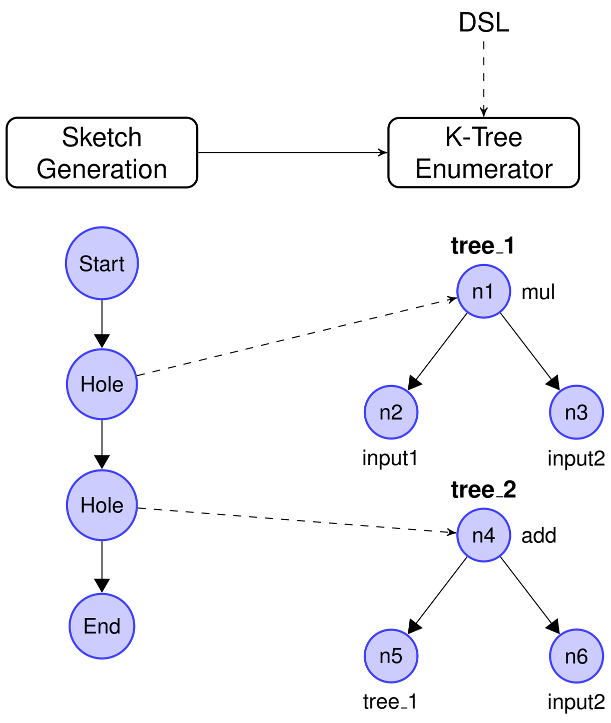

Figure 4.14: Enumeration of k-trees for a given sketch with two holes to fill.

satisfies that formula represents the assignment of a symbol of the given DSL to each node in the trees. Within the possible tree encoding approaches, we have selected the line-based encoding.

In the line-based encoding, a program is represented using a sequence of trees of depth 1, where each tree represents one operation of the program, as in an imperative language. PUFS uses an adaptation of the line-based encoding presented by Orvalho et. al. [21]. A program representing a flow with Assign nodes only can be seen as a sequence of operations, therefore, we want to fill each hole using a k-tree rather than one single k-tree to represent the whole program, as illustrated in Figure 4.14. However, when it comes to flows with If nodes that does not apply, and for that same reason we use an adaptation of the encoding instead of the original one. Furthermore, the trees are enumerated in increasing depth until a solution is found or until a timeout is reached.

In the following sections we describe the variables and constraints used to encode the line-based tree representation used in our synthesizer.

4.3.3 Line-based Encoding with Conditionals

When considering flows with Assign nodes only, the previous encoding, described in section 3.3, would be enough, since a program, with assignment expressions only, can be seen as program written in an imperative language where each line would be the expression associated to each Assign node. However, in the presence of If nodes, the structure of our programs is not so straight forward. Therefore, in order to synthesize flows with conditional expressions we have implemented the following constraints in addition to the ones presented in section 3.3.1.

Recall that D is the DSL, $Prod(D)$ the set of production rules in D and $Term(D)$ the set of terminal symbols in D. Furthermore, $Types(D)$ represents the set of types used in D and $Type(s)$ the type of symbol $s \in Prod(D) \cup Term(D)$. If $s \in Prod(D)$, then $Type(s)$ corresponds to the return type of production rule s.

Consider Σ the set of symbols used in the program. Besides the production rules and terminal symbols, there is one additional symbol ret for each line in the program. Let $Ret = \{ret_i : 1 \leq i \leq n\}$ represent the set of return symbols in the program, then $\Sigma = Prod(D) \cup Term(D) \cup Ret$.

Furthermore, 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 operations in the DSL have a smaller arity than k, the empty symbol ϵ is introduced, so that every leaf node has an assigned symbol. We assume $id(\epsilon) = 0$.

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

- $O = \{op_i : 1 \leq i \leq n\}$: each variable op_i represents the production rule used in line i.
- $T = \{t_i : 1 \leq i \leq n\}$: each variable t_i represents the return type of the expression in line 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 in line i.

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

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

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

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

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

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

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

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

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

However, if the sketch to be completed has If nodes, there will be more than a single execution path, so, the results of an operation can only be used within the following operations of the same execution branch. Therefore, we have the following constraint. Let $PreviousHoles(i)$ be the set of lines used in previous holes from the same execution path as node i, excluding lines that are used to fill If nodes.

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

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

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

A terminal symbol $t \in Term(D)$ cannot be used as argument j of an operation 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 operation i can be smaller than k, in that case, the empty symbol ϵ is assigned to the arguments above the productions 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 which 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. Consider L the set of all lines corresponding to nodes that point to an End node. Since the last line of a program corresponds to the program's output, the operation of each one of the lines in L must be one the productions in P_{out}.

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

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

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

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

$$\forall ret_r \in Ret(D) : \left(\sum_{r < i \leq n, 1 \leq j \leq k} (a_{ij} = id(ret_r)) \right) >= 1 \tag{4.11}$$

4.3.4 Decider

Once the sketch is filled with the corresponding expressions in each node, the decider evaluates if the resulting program satisfies the specification. To perform this evaluation we developed a flow interpreter, which takes a graph that represents a flow and interprets the resulting program using the input values to obtain the corresponding outputs of that program. If, for every input, the program returns the corresponding output from the specification the decider considers that a solution was found. Otherwise, a new sketch is generated to be filled by the synthesizer. This process is repeated until a solution is found or until a time limit is reached.

Chapter 5

Evaluation

This chapter evaluates the performance of the synthesizer presented in Chapter 4. The synthesizer is evaluated in terms of its running time and number of instances solved. We provide a brief description of the benchmarks provided, followed by a description of the pre-selection and categorization of the benchmarks in Section 5.1. Finally, Section 5.2 provides the evaluation results and the corresponding analysis.

5.1 Examples Generation

In order to evaluate our PBE-based synthesizer, we need to get a set of benchmarks in the OutSystems platform that represent the type of functions we intend to synthesize. The set of examples we use consist of a set of real-world examples of pure functions implemented the OutSystems Platform.

As mentioned before, the main goal is to facilitate the implementation of pure functions, which are functions that have no side effects and their outputs are determined by its inputs, similarly to other types of functions. For the purpose of this thesis, we will only consider functions consisting of just Assign and If nodes.

However, this set of examples needs to go through a pre-selection process so that our set of examples consists only of supported examples by the current DSL. The pre-selection process is further described in subsection 5.1.1. Finally, once the examples are pre-selected and categorized we use an interpreter (subsection 4.3.4) to sample the input-output examples to be used as specification for our synthesizer.

5.1.1 Examples Pre-selection

The examples consist of a set of flows that only have nodes of type Start, End, Assign and If. However, the examples must fulfill some requirements in order to be useful examples, so we needed to perform some categorization in order to select the appropriate ones to represent the cases we wish to synthesize.

The mentioned requirements are the following:

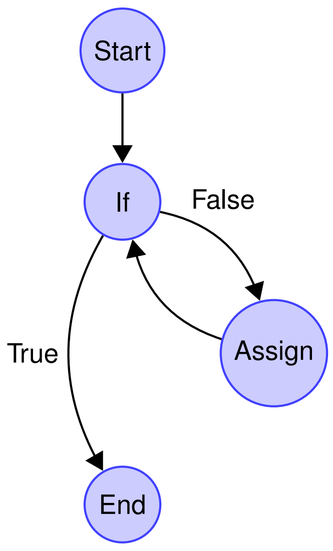

Figure 5.1: Flow with loop

Table 5.1: Total for each pre-selection category and overlapping of each category

Pre-selection category	# Instances
Initial number of flows	13959
Loops	861
Out-of-scope variable types	8825
Out-of-scope functions	4655
Loops and out-of-scope variables types	703
Loops and out-of-scope functions	275
Out-of-scope variables types and out-of-scope functions	2974
Loops and out-of-scope variables types and out-of-scope functions	249
Total number of flows after pre-selection	3321

- No loops. Even though these flows do not have Loop type nodes there can still be loops using If nodes, as shown in figure 5.1.

- All function calls that appear in the flow are calls to built-in functions. Besides filtering flows with loops, we also found that some flows were using variables whose type did not belong to the variables basic types available in the platform along side with the use of functions that were not the builtin functions provided by the platform, but functions developed by the user. Given that, these flows will not be used as examples for the synthesizer.

- All variables that appear in the flow are of the basic types supported by the synthesizer: integer, decimal, text and boolean.

In order to develop a better understanding of the obtained benchmarks, table 5.1 presents the number of instances for each one of the pre-selection categories along with the number of instances that overlaps between each of them. Also, the Venn diagram in figure 5.2 provides a visual representation of the exact

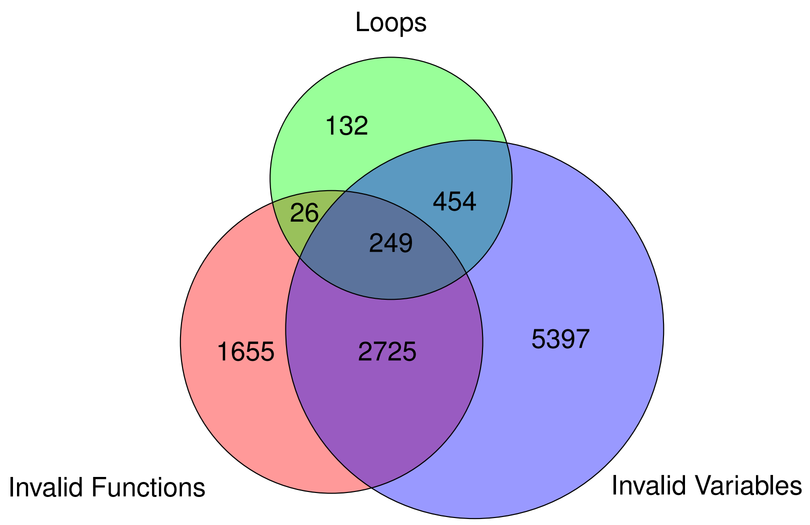

Figure 5.2: Venn Diagram for Pre-selection of Examples

number of instances for each of the overlapping categories along side the number of flows for each exclusive category.

5.1.2 Examples Categorization

Given that there exists significant variety within the selected instances, it is important to perform some sort of categorization in order to better evaluate the synthesizer, and also to be able to have a better sense of the main characteristics of the data being used.

The flows were categorized using 3 characteristics:

- Size of the flow.
- Flows with/without If nodes.
- Flows with the Built-In Types and Built-In Functions supported by the synthesizer.

The first category is based on the size of the flows, i.e., the flows are categorized based on the number of nodes. This is important, given that, the greater the number of nodes in a flow, the more operations it can perform, and, consequently the search space will also be more extensive. The sizes go from 3 to 187 nodes. The plot in figure 5.3 provides a more descriptive view of all the sizes found and the number of instances for each one of the sizes.

The second category separates the flows based on the presence of If nodes, i.e., flows containing If nodes are separated from those that do not. The rationale behind this categorization is that programs with control structures are expected to be much harder to synthesize due to the introduction of branching. Table 5.2 provides a more descriptive view of the number of instances with If nodes and without non-If nodes and among those, which are considered within the scope of our synthesizer, meaning that these instances only use built-in types and built-in functions supported by the platform.

It is possible to verify that there is a significant difference between the total number of valid instances and the total number of instances in general. This difference corresponds to the excluded instances cor-

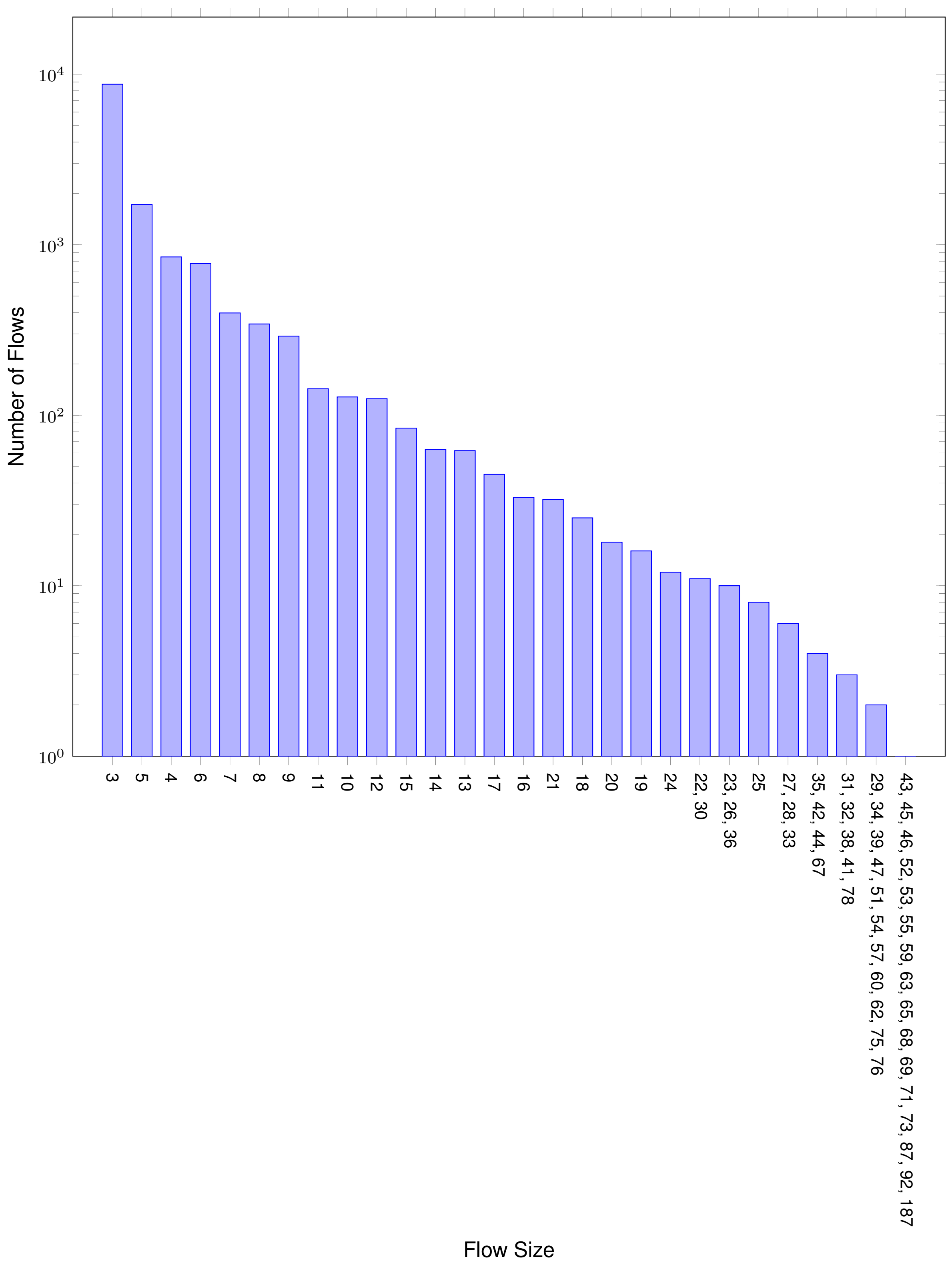

Figure 5.3: Number of instances for each size category

Table 5.2: Number of instances for each category of flows with conditional nodes and without conditional nodes

	# Instances	# Valid Instances	Total
Non-conditionals	9454	2217	11671
Conditionals	4505	1104	5609
Total	13959	3321	

Table 5.3: Number of instances that are supported by the synthesizer for each category of flows with conditional nodes and without conditional nodes

	# Supported Instances	# Not Supported Instances	Total
Non-conditionals	995	1222	2217
Conditionals	471	633	1104
Total	1466	1855	

responding to flows that contain functions in their expressions that do not correspond to built-in functions of the platform, i.e., the functions correspond to functions created by the users and used in the flows and as such they are beyond the scope of our synthesizer.

The third and final category contains the flows that are supported by the current DSL being used by the synthesizer. As it was mentioned in section 4.1, the OutSystems platform provides a set of built-in types[1] and built-in functions[2] such as mathematical computations, data transformations, among others, which can be used within OutSystems expressions. However, the synthesizer does not support all of them, for now, it supports the Integer, Decimal, Text and Boolean built-in types and the set of Math Built-In Functions, Numeric Built-In Functions and some of the Text Built-In Functions, described previously in table 4.1. Table 5.3 provides a view over the total number of flows, once the selection of instances supported by the current DSL is done, for instances having conditional nodes and instances that only have assignment nodes.

We noticed that, among the selected benchmarks, there exist several repeated instances, i.e., several pure functions with exact same behavior/semantics. Assume we have 40 instances that calculate the length of a string given as input. Once we evaluate our synthesizer, those repeated instances would induce bias in the results of the evaluation. Therefore, we have performed one last selection performed manually consisting of removing repeated instances among the examples. Table 5.4 provides the final counting of the instances used for evaluating the synthesizer corresponding to a total of 566 instances.

5.1.3 Generation of input-output examples

As mentioned in the previous section, the examples of flows available to test our synthesizer consist of real world examples of flows in the OutSystems platform corresponding to pure functions, developed by

[1]https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Data_Types/Available_Data_Types

[2]https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic/Built-in_Functions

Table 5.4: Total number of instances that are supported by the synthesizer without repeated instances

	Non-conditionals	Conditionals	Total
# Supported Instances	309	257	566

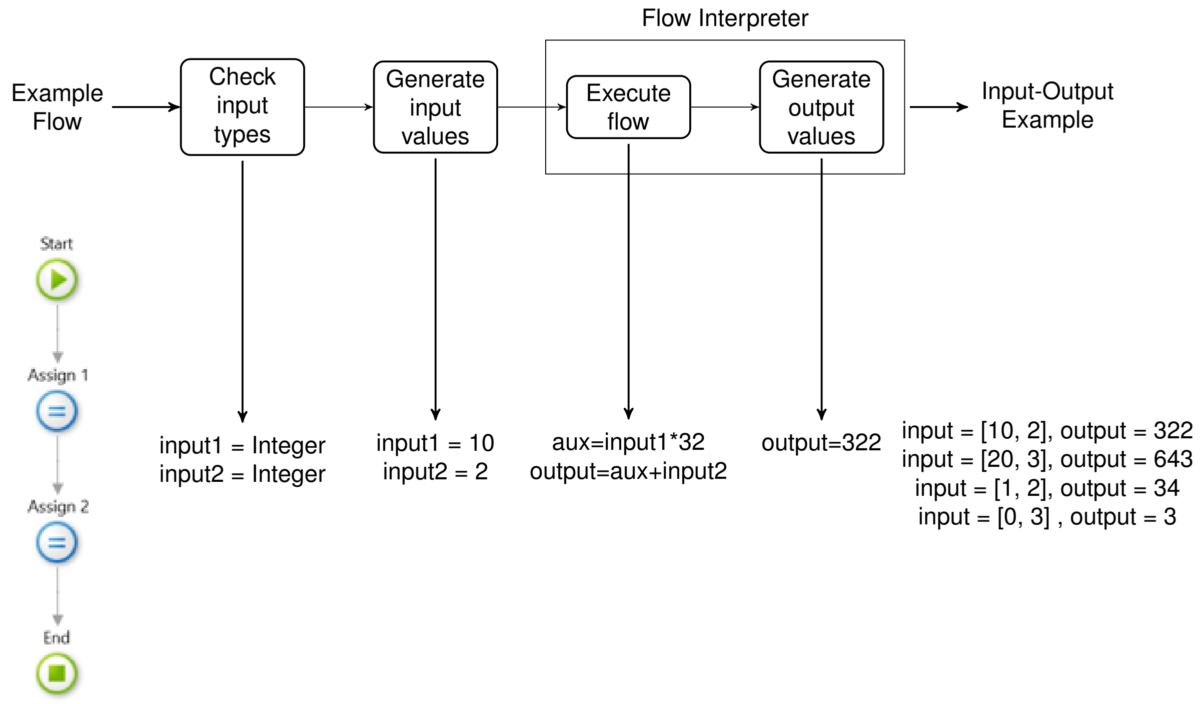

Figure 5.4: Example flows interpreter.

users. However, we need input-output examples and not the flows, so we need to execute the flows in order to obtain the corresponding input-output pairs.

In order to to execute the flows we used the flow interpreter from section 4.3.4, which takes an action flow in an intermediate representation, and then computes the respective output in order to generate the input-output examples. The process of generating these instances has 4 stages, as illustrated in figure 5.4:

1. Check input types
2. Generation of input values
3. Execution of the flow
4. Generation of output values

Check input types and generation of the input values Upon receiving an example flow, the interpreter checks the input variables and its types, so that it generates random values for each variable according to its type.

Execution of the flow and generation of output values Once the values are generated, the interpreter takes the inputs and interprets the program in order to obtain the corresponding output values.

This approach allows to perform the generation of examples for most cases, however for instances that should cover edge cases, this generation of examples was performed manually in order to obtain the correct solution in terms of matching the user's intent.

5.2 Experimental Results

The goal is to evaluate how many pure functions PUFS is able to synthesize and how quickly. We also evaluate if providing additional information, such as constants to be used in the program, has a significant impact in the performance. In this section we provide the results of the evaluation for instances using assignments only. Although we present the encoding to synthesize flows containing If nodes in Section 4.3.3, we were not able to retrieve the results for instances that use conditional nodes.

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

5.2.1 Evaluation

We want to evaluate our synthesizer in terms of runtime and how many instances we can solve for a given time frame. Additionally, we evaluate the impact of providing constants along with the input-output examples.

We present results for the synthesizer presented in Chapter 4 with the following configurations. For each instance we ran the synthesizer with 5 input-output examples. For instances in which the solution is expected to use constants we provided the corresponding constants in the specification file along with the input-output examples. These constants consist of integer, decimal and string values. We ran the synthesizer for instances with constants and with no constants provided. This is due to the fact that the synthesizer is not able to synthesize programs with constants for now, so solutions that require constants would not be found using the present configuration unless these were provided as one of the input values.

Table 5.5 shows the results of both approaches in terms of the number of instances solved and limit time considered. We consider an instance solved if the synthesized program satisfies the input-output examples. However, even though it is considered solved it does not mean it is correct, i.e. it might not match the intended solution. Matching the user intent means that the solution must satisfy the specification as well as capturing the user intent (this verification is performed manually). For this same reason some of the instances example values had to be manually generated in order to obtain the most

Table 5.5: Comparison of number of instances solved for the different timeout values.

Timeout(s)	60	120
Without constants	83	115
With constants	103	121

correct solution possible, as it was mentioned in 5.1.3.

5.2.2 Discussion

From the presented results in plot 5.5, we can verify that as the complexity of the program increases so does the time to find solution. As the enumeration process is done in an ascending order of size of the sketch, the bigger the sketch we want to complete the longer it will take to complete it and find a solution that satisfies the specifications. We need to take into account that the bigger the size of the program we are synthesizing, the longer it will take for the sketch completion stage to be completed, due to the fact that the number of holes we wish to fill increases the number of expressions to be synthesized.

As each hole is filled with an expression in the form of a k-tree and as these are also enumerated in ascending complexity, in terms of depth, the enumeration process becomes more complex and more time consuming due to the fact that as the program space increases so does the time to perform the search over the space in order to find the desired program. Therefore the use of pruning techniques can be an very advantageous addition to the solution in order to reduce the space of programs and find a solution in a smaller time frame.

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

Regarding the use of constants, from plot 5.5 and table 5.5, we can verify that the use of constants has a significant impact in the number of solved instances, which is more evident in instances solved under 1 minute. This impact would be expected due to the fact that the synthesizer does not enumerate candidate programs containing constants, unless these are provided by the user. However, the addition of constants also increases the time it takes to find a solution, due to the fact that more candidate programs are enumerated. Which would explain the results for the timeout of 2 minutes, illustrated in plot 5.5, where we verify that the number of solved instances without constants is almost the same as when providing the constants. For this particular case, it is important to evaluate if the trade-off between the additional time spent enumerating more programs, compensates by finding a solution in a reasonable time.

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

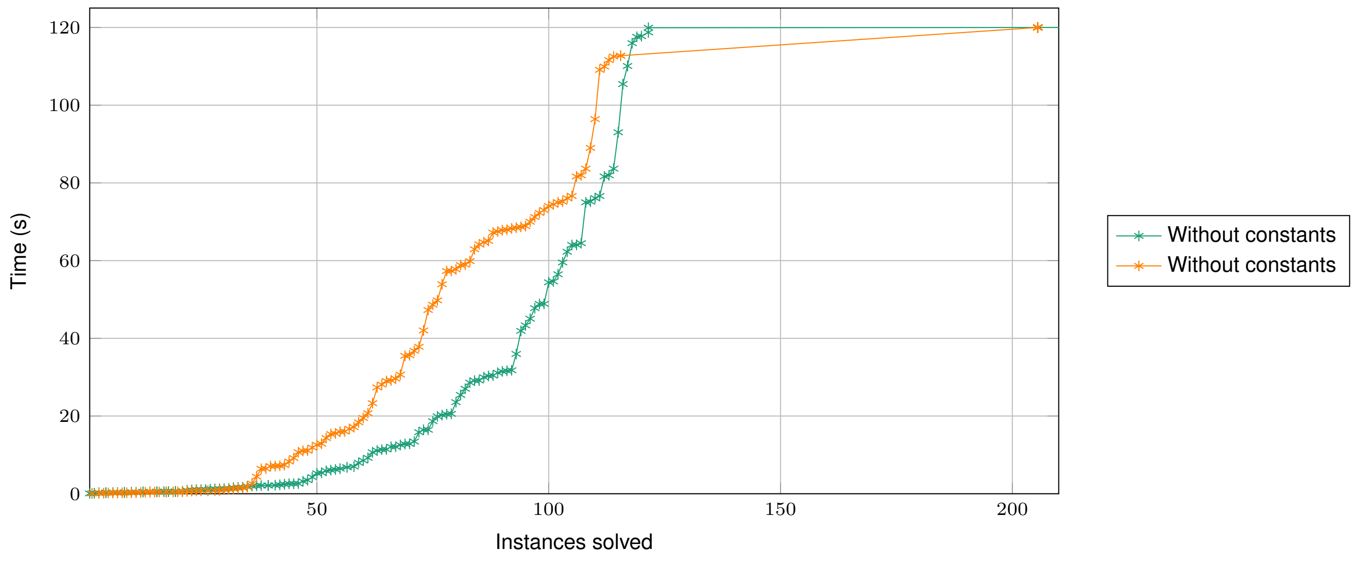

Figure 5.5: Number of instances solved throughout time for a timeout of 120 seconds.

Finally, in chapter 4, we have presented a proposal of an adaptation of the encoding developed by Orvalho et. al. [21], however we were not able to retrieve results for that encoding, and for that reason those results are not provided in this chapter as mentioned in the beginning.

Chapter 6

Conclusions and Future Work

In this thesis, we tackle the problem of synthesizing pure functions from examples in the OutSystems platform. We focus on functions that manipulate integers, decimals, text and Booleans. The OutSystems platform main goal is to provide an easier and faster experience in development and integration of web and mobile applications. This platform gives users, with no programming background, a tool that allows them to develop an application with no need for specific knowledge, and also provides a faster and automated approach to users with more specialized knowledge. Therefore, it is in our interest to provide a simple approach for the user to be able to generate the pure functions with only a small number of examples in just one click. These type of functions come across very often, in the form of data wrangling tasks, among others. Having such a repetitive type of task might become tedious and add more complexity to the overall tasks, which leads to a need of automating this type of functions.

In this dissertation we presented a novel approach to synthesize pure functions in the form of flows in the OutSystems platform, from a set of input-output examples. We survey the state of the art in program synthesis and implemented PUFS, a PBE-based pure function synthesizer. The synthesizer employs the use of sketches as the underlying structure of our programs and enumerative search, where SMT is used to search the program space. We tested PUFS in a set of real-world examples of pure-functions developed in the OutSystems platform, from which the results of our experiments revealed we are able to synthesize 33% of the benchmarks within less then a minute. However, the results also revealed that in a significant amount of examples we were not able to find a solution within a limited amount of time due to the dimension of the program space or due to an incomplete specification.

Given the experimental results for flows with assignments only, we believe the current solution could benefit from some pruning techniques, in order to reduce the search space and possibly overcome the time limitation to find a solution. As follow up work, it would be important to verify the results for the adaptations of the encoding introduced in Chapter 4 to evaluate if the approach provides favorable results.

Finally, it would also be interesting to use a ranking technique as an ambiguity resolution technique in order to guide the synthesizer into finding a program that is more likely to lead to a solution that satisfies both the specification and corresponds to the user intent. Moreover, it is important to explore if this approach scales if the DSL was to be extended, given that it would be interesting to extend the current DSL to support more data types and more functions. Besides the DSL it would be particularly interesting to support other type of nodes that allow other types of operations.

Bibliography

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

[2] Y. Chen, R. Martins, and Y. Feng. Maximal multi-layer specification synthesis. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 602–612. ACM, 2019. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338951. URL `https://dl.acm.org/citation.cfm?id=3338906`.

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

[4] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL `https://doi.org/10.1007/978-3-540-78800-3_24`.

[5] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In L. K. Dillon, W. Visser, and L. A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 345–356. ACM, 2016. doi: 10.1145/2884781.2884786. URL `https://doi.org/10.1145/2884781.2884786`.

[6] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017. URL `http://proceedings.mlr.press/v70/devlin17a.html`.

[7] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In Cohen and Vechev [3], pages 422–436. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062351.

[8] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017. ISBN 978-1-4503-4660-3. URL `http://dl.acm.org/citation.cfm?id=3009851`.

[9] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In J. S. Foster and D. Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435. ACM, 2018. doi: 10.1145/3192366.3192382.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL `http://dl.acm.org/citation.cfm?id=1926385`.

[11] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, editors, *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012*, pages 8–14. IEEE Computer Society, 2012. doi: 10.1109/SYNASC.2012.69. URL `https://doi.org/10.1109/SYNASC.2012.69`.

[12] S. Gulwani. Programming by examples: applications, algorithms, and ambiguity resolution. In W. Vanhoof and B. Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, page 2. ACM, 2017. ISBN 978-1-4503-5291-8. doi: 10.1145/3131851.3131853. URL `https://doi.org/10.1145/3131851.3131853`.

[13] S. Gulwani and P. Jain. Programming by examples: PL meets ML. In B. E. Chang, editor, *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, volume 10695 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017. doi: 10.1007/978-3-319-71237-6_1. URL `https://doi.org/10.1007/978-3-319-71237-6_1`.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506.

[15] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010.

[16] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010. ISBN 978-1-60558-719-6.

[17] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12):1914–1917, 2019. doi: 10.14778/3352063.3352098. URL `http://www.vldb.org/pvldb/vol12/p1914-martins.pdf`.

[18] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In C. Latulipe, B. Hartmann, and T. Grossman, editors, *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, pages 291–301. ACM, 2015. doi: 10.1145/2807442.2807459. URL `https://doi.org/10.1145/2807442.2807459`.

[19] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine. Neural sketch learning for conditional program generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=HkfXMz-Ab`.

[20] M. I. Nye, L. B. Hewitt, J. B. Tenenbaum, and A. Solar-Lezama. Learning to infer program sketches. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR, 2019. URL `http://proceedings.mlr.press/v97/nye19a.html`.

[21] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In T. Schiex and S. de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2019. doi: 10.1007/978-3-030-30048-7_34. URL `https://doi.org/10.1007/978-3-030-30048-7_34`.

[22] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. SQUARES : A SQL synthesizer using query reverse engineering. *Proc. VLDB Endow.*, 13(12):2853–2856, 2020. URL `http://www.vldb.org/pvldb/vol13/p2853-orvalho.pdf`.

[23] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of*

SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 107–126. ACM, 2015. ISBN 978-1-4503-3689-5. URL `http://dl.acm.org/citation.cfm?id=2814270`.

[24] K. Shi, J. Steinhardt, and P. Liang. Frangel: Component-based synthesis with control structures. *CoRR*, abs/1811.05175, 2018. URL `http://arxiv.org/abs/1811.05175`.

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

[26] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2015. ISBN 978-3-319-21689-8.

[27] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[28] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065045.

[29] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006. ISBN 1-59593-451-0.

[30] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In Cohen and Vechev [3], pages 452–466. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.

[31] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett. Sketch-driven regular expression generation from natural language and examples. *Trans. Assoc. Comput. Linguistics*, 8:679–694, 2020. URL `https://transacl.org/ojs/index.php/tacl/article/view/2135`.

[32] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In E. Denney, T. Bultan, and A. Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 224–234. IEEE, 2013. URL `https://ieeexplore.ieee.org/xpl/conhome/6684409/proceeding`.