TÉCNICO LISBOA

FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions

Margarida de Almeida Cruz Ferreira

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 Paolo Romano
Supervisor: Professor Maria Inês Camarate de Campos Lynce de Faria
Member of the Committee: Professor Alexandra Sofia Ferreira Mendes

November, 2020

“DON’T PANIC”

– Douglas Adams,
The Hitchhiker’s Guide to the Galaxy

Acknowledgments

First, I want to extend my gratitude to Professor Inês Lynce for her unwavering support, for encouraging me to pursue a career in research and for being an outstanding role model. Her invaluable advice has helped me not only during the development of this dissertation, but continuously over the past few years.

I would like to thank Miguel Neves for his patient explanations, for his very thorough (and very colourful!) text revisions, for always knowing a paper about mostly everything, and for offering me the first "queijada da Graciosa" I ever tried.

I want to thank Miguel Ventura for welcoming me at OutSystems and for sharing his knowledge on regular expressions. I hope when he has a problem and he decides to solve it with a regular expression, he is left with $n < 2$ problems now!

I am very grateful to Ruben Martins for sharing his expertise in program synthesis and for his advice, for always asking the questions to which I do not have the answers (yet!) and for helping me write in-depth results sections.

I would also like to thank Professor Vasco Manquinho, for his remarkable ability to keep the SAT group's servers running, despite our continuous overusing them.

I would like to thank Ricardo and Catarina for facing this challenge with me. We often came across similar troubles and I have to thank them, above all, for the many uncontrollable fits of laughter we shared, which usually fix them all. I'm also grateful to Daniel, one of my oldest friends, and Pedro, one of the newest, for sharing their experiences, troubles and successes. They strongly inspire my own.

I would like to thank Vik, Katla and Corisco, my feline companions, always willing to help me chase away the bugs. I want to thank my parents and my sister, for encouraging me, for supporting me, and for patiently waiting until I hand in this dissertation before I go back to continually fixing their email clients and smartphone settings. They are heroes to everyone, but especially to me.

This work was supported by OutSystems, by national funds through FCT, under project UIDB/50021/2020, and project ANI 045917 funded by FEDER and FCT.

Resumo

Os formulários digitais são um método popular para recolha de dados. O suporte para validações em tempo real que filtram dados inválidos torna-os particularmente desejáveis. Validações baseadas em expressões regulares são usadas frequentemente em formulários digitais para evitar que os utilizadores introduzam dados no formato errado. No entanto, a escrita destas validações pode representar um desafio para alguns utilizadores.

Neste documento, apresentamos o FOREST, um sintetizador de expressões regulares para validação de formulários digitais. O FOREST produz uma expressão regular que corresponde ao padrão desejado para os valores de *input*, um conjunto de grupos de captura que permitem extrair deles mais informação, e um conjunto de condições sobre grupos de captura que garantem a validade de valores inteiros no *input*. O nosso método de síntese é baseado em procura enumerativa e usa um *solver* de Satisfazibilidade Módulo Teorias (SMT) para explorar e podar o espaço de procura. Propomos uma nova representação para a síntese de expressões regulares, multi-árvore, que induz padrões nos exemplos e os usa para dividir o problema através de uma abordagem dividir para conquistar. Apresentamos ainda uma nova codificação SMT para sintetizar as condições sobre capturas para uma dada expressão regular. Para aumentar a confiança na expressão regular sintetizada, implementamos um modelo de interação com o utilizador com base em *inputs* distintivos.

Avaliámos o FOREST em instâncias de validação de formulários do mundo real com base em expressões regulares. Os resultados experimentais mostram que o FOREST retorna com sucesso a expressão regular desejada em 74% das instâncias e supera o REGEL, um sintetizador de expressões regulares estado-da-arte.

Palavras-chave: Síntese de programas, Programação-por-Exemplo, Satisfazibilidade Módulo Teorias, Expressões Regulares, Validação de Formulários.

Abstract

Digital forms are a popular method for collecting data. The support for real-time validations that filter invalid provided data makes them particularly desirable. Form validators based on regular expressions are often used on digital forms to prevent users from inserting data in the wrong format. However, writing these validators can pose a challenge to some users.

In this document, we present FOREST, a regular expression synthesizer for digital form validations. FOREST produces a regular expression that matches the desired pattern for the input values, a set of capturing groups that extract some information from them, and a set of conditions over capturing groups that ensure the validity of integer values in the input. Our synthesis procedure is based on enumerative search and uses a Satisfiability Modulo Theories (SMT) solver to explore and prune the search space. We propose a novel representation for regular expressions synthesis, multi-tree, which induces patterns in the examples and uses them to split the problem through a divide-and-conquer approach. We also present a new SMT encoding to synthesise capture conditions for a given regular expression. To increase confidence in the synthesised regular expression, we implement a user interaction model based on distinguishing inputs.

We evaluated FOREST on real-world form-validation instances using regular expressions. Experimental results show that FOREST successfully returns the desired regular expression in 74% of the instances and outperforms REGEL, a state-of-the-art regular expression synthesizer.

Keywords: Program Synthesis, Programming by Example, Satisfiability Modulo Theories, Regular Expressions, Form Validation.

Contents

List of Figures

List of Tables

List of Acronyms

SAT Propositional Satisfiability

CNF Conjunctive Normal Form

MaxSAT Maximum Satisfiability

SMT Satisfiability Modulo Theories

LIA Linear Integer Arithmetic

EUF Equality with Uninterpreted Functions

MaxSMT Maximum Satisfiability Modulo Theories

PBE Programming by Example

DSL Domain Specific Language

CFG Context-Free Grammar

CEGIS Counterexample Guided Inductive Synthesis

OGIS Oracle Guided Inductive Synthesis

DFS Depth-First Search

Chapter 1

Introduction

As computers steadily become vital tools in our day-to-day lives, the need arises to extend the ability to develop computer applications to ampler audiences, including those who lack a programming background. OutSystems [48] is a low-code development platform that enables its users to build applications through a graphical interface and integrate them with existing systems. It speeds up the development of web and mobile applications, at the same time making the task more accessible to users from different backgrounds.

Data analysis is a powerful resource with various applications that has flourished as a research field over the years. Digital forms are increasingly popular tools for structured data collection. They can gather large amounts of data from users all across the globe in a short amount of time. Furthermore, in digital forms, we can perform real-time validations on the input fields. Well-written validations result in overall cleaner and standardised data, free of invalid values, such as typographical mistakes ('typos') and format inconsistencies, which requires less processing to ready for analysis.

Regular expressions (also known as regexes) are powerful mechanisms for describing patterns in text with numerous applications. One notable use of regular expressions is precisely to perform form validations on the input fields of digital forms. Aside from validating the format of form input strings, regular expressions can be coupled with capturing groups. A capturing group is a sub-regex within a regex that is indicated with parenthesis and captures the text matched by the sub-regex inside them. Capturing groups are used to extract information from text and, in the domain of form validation, they can be used to enforce conditions over values in the input string.

One of the many features offered by OutSystems is the automatic generation of forms based on a high-level description of desired fields, to which users can subsequently add custom hand-written validations that are checked before the form is submitted. Its usefulness notwithstanding, form validations often rely on complex regular expressions which require programming skills that not all users possess. Furthermore, hand-written validations are error-prone. Programmers often miss corner cases, admitting less-than-obvious mistakes that more imaginative form-fillers may commit.

Program synthesis is the task of automatically generating a program that satisfies some desired behaviour expressed as a high-level specification. Since low-code platforms have the ultimate vision

of making application development accessible to everyone, there is a lot of potential in the integration of program synthesis in such platforms. Owing to its versatility and usefulness, program synthesis has recently attracted interest from various research communities (e.g. constraint solving [1, 8, 30, 45, 54, 73], programming languages [15, 17, 70], machine learning [10, 14, 41], and deep learning [5, 8, 27, 49, 53]), leading to various promising proposals.

To help users write regular expressions, prior work has proposed to synthesise regular expressions from natural language [8, 31, 34, 73, 74] or from positive and negative examples [8, 22, 32, 72]. Even though these techniques assist users in writing regular expressions for search and replace operations, they do not specifically target digital form validation and do not take advantage of the structured format of the data.

1.1 Motivating Example

In this thesis, we propose FOREST, a new program synthesizer for regular expressions that targets digital form validations. FOREST takes as input a set of examples and returns a regex validation that validates them. FOREST accepts three types of examples:

1. **valid examples**: correct values for the input field. The valid examples can be accompanied by **captures**, substrings of the examples that capture relevant information;
2. **invalid examples**: incorrect values for the input field due to their *format*, and
3. **conditional invalid examples** (optional): incorrect values for the input field due not to their format but to their *values*.

FOREST outputs a regex validation, consisting of three components:

1. a **regular expression** that matches all valid and none of the invalid examples and
2. **capturing groups** that reflect the captures provided with the valid examples;
3. **capture conditions** that express integer conditions for values in the examples that are satisfied by all valid but none of the conditional invalid examples.

Suppose a user is writing a form where one of the fields is a date that must respect the format DD/MM/YYYY. The user wants to accept the input strings (valid examples):

19/08/1996	22/09/2000	29/09/2003
26/10/1998	01/12/2001	31/08/2015

But not (invalid examples):

19/08/96	22.09.2000	29/9/2003
26-10-1998	1/12/2001	2015/08/31

A regular expression can be used to enforce this format. However, if the user is not proficient in the usage of this formalism, writing it can be a challenging task. Even if the user is familiar with regular expressions, writing a large number of such validations for all the fields of a form can become monotonous and error-prone. Instead, the user may simply use the two sets of examples as input to FOREST, who outputs the regular expression `[0-9]{2}/[0-9]{2}/[0-9]{4}`.

Suppose the user wants to validate not only the format, but also the values in the date. We consider as conditional invalid:

33/08/1996	22/13/2000	12/31/2003
26/00/1998	00/12/2001	52/03/2015

To ensure only valid values are inserted as the day and month, we can add capturing groups to our regular expression. A capturing group is represented by a set of parenthesis. They capture the text matched by the regex inside them, that can later be used with a numbered reference. In this document we use the notation $\$i, i \in 0, 1, ...$ to reference the integer value captured by the $(i+1)^{\text{th}}$ group. Then, conditions can be applied to these values. In this situation, the desired validation (and the output of FOREST) includes the same regular expression as before, now extended with two capturing groups and four integer conditions over them:

$$\overbrace{\underbrace{\texttt{([0-9]\{2\})}}_{\text{capturing group}}\texttt{/}\underbrace{\texttt{([0-9]\{2\})}}_{\text{capturing group}}\texttt{/[0-9]\{4\}}}^{\text{regex}}, \underbrace{\$0 \leq 31 \wedge \$0 \geq 1 \wedge \$1 \leq 12 \wedge \$1 \geq 1}_{\text{capture conditions}}$$

Finally, suppose the user wishes not only to validate the input string but also to extract some information from it. Still using the dates example, the user could wish to extract the year from each date, so it could be used afterwards. Thus, the user could extend the valid example strings with captures:

19/08/1996, 1996	22/09/2000, 2000	29/09/2003, 2003
26/10/1998, 1998	01/12/2001, 2001	31/08/2015, 2015

Using a capture group, we can extract the year from each date. If the user provides these example captures alongside the valid examples, FOREST outputs the regular expression complete with the capture group necessary to extract them: `[0-9]{2}/[0-9]{2}/([0-9]{4})`. The captured text corresponds to the desired information.

1.2 Contributions

As we can see in the motivating example, data inserted into digital forms is usually structured and shares a common pattern among the valid examples. In this example, the data has the shape `dd/dd/dddd` where `d` corresponds to a digit. FOREST takes advantage of this structure by automatically detecting these patterns and using a divide-and-conquer approach to split the expression into simpler sub-expressions,

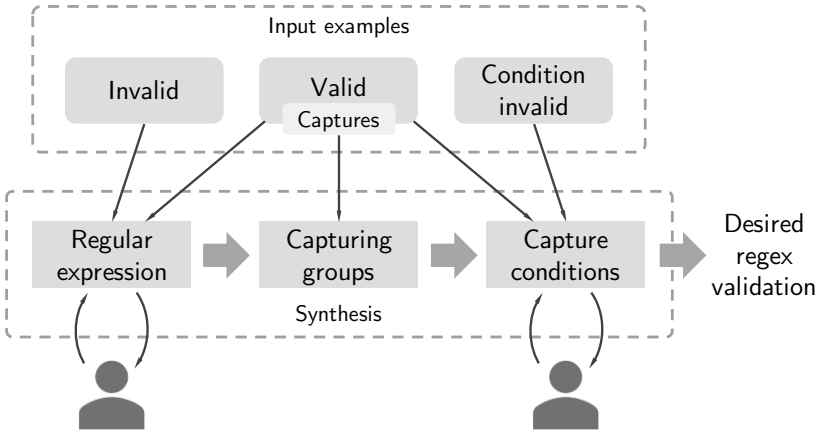


Figure 1.1: FOREST's regex synthesis pipeline

solving them independently, and then merging their information to obtain the final regular expression. The obtained regular expression validates the form input field's format. Additionally, FOREST computes a set of capturing groups over the regular expression to extract information, and integer conditions over captured values that further constrain the accepted inputs for that field.

Input-output examples do not require specialised knowledge and are accessible to users. However, there is one downside to using examples as a specification: they are ambiguous. There can be solutions that, despite matching the examples, do not produce the desired behaviour in situations not covered in them. The ambiguity of input-output examples raises the necessity of selecting one among multiple candidate solutions. To this end, we incorporate a user interaction model based on distinguishing inputs for both the synthesis of the regular expression and the synthesis of the capture conditions.

Our synthesis procedure is split into three stages, each relative to an output component. First, FOREST synthesises the regular expression, which is the basis for the synthesis of capturing groups. Secondly, FOREST computes a set of capturing groups that match the captures provided by the used alongside the valid examples. If it is impossible to compute correct capturing groups using the current regular expression, FOREST reverts to the first stage to produce another regular expression. Finally, FOREST synthesises the capture conditions, by first computing a new set of capturing groups and then the conditions to be applied to the resulting captures. Again, if it is not possible to compute correct conditions with the current regular expression, FOREST revers to the first stage to try to synthesise another regular expression. Figure 1.1 shows the regex validation synthesis pipeline.

All three stages of our synthesis algorithm employ enumerative search, a common approach to solve the problem of program synthesis [15, 17, 32, 45, 58]. To circumvent the ambiguity of input-output examples, FOREST implements an interaction model. A new component, the *distinguisher*, ascertains, for any two given programs, whether they are equivalent. When FOREST finds two different validations that satisfy all examples and are not equivalent, it creates a *distinguishing input*: a new input that has a different output for each solution. To disambiguate between two programs, FOREST shows the new input to the user, who classifies it as valid or invalid, effectively choosing one program over the other. The new input-output pair is added to the examples, and the enumeration process continues until there is only one solution left.

In summary, this dissertation makes the following contributions:

- We propose a multi-tree SMT representation for regular expressions that leverages the structure of the input examples to apply a divide-and-conquer approach.
- We propose several techniques to prune the search space, based on regex properties.
- We propose a new method to synthesise capturing groups for a given regular expression and integer conditions over the resulting captures.
- We implemented our approach in a tool, FOREST, that interacts with the user to disambiguate the provided specification. We evaluate FOREST on real-world form-validation instances that use regular expressions. Experimental results show that FOREST can synthesise 74% of the regex validations that match the user intent, and that FOREST outperforms REGEL, a general state-of-the-art synthesizer for regular expressions.

1.3 Document Structure

This document is organised as follows.

We begin in Chapter 2 by introducing some background concepts used throughout the rest of the document. We introduce the concepts of regular language and regular expression, the logic concepts used throughout this document, and the main components that characterise a program synthesizer. Chapter 3 presents a survey of program synthesis algorithms with emphasis on regular expression synthesis. We look into *AlphaRegex* and REGEL, two regular expression synthesizers that propose optimisations specific to that domain.

We proceed to Chapter 4, where we take a look into how FOREST synthesises the first part of its regex validation: a regular expression. In Chapter 5 we show how FOREST synthesises the second and third parts of the regex validation: a set of capturing groups that reflect the user-provided captures, and a set of integer conditions over capturing groups that invalidate the conditional invalid examples

We move on to Chapter 6, where we present and discuss our experiments. We start by comparing FOREST to REGEL, a state-of-the-art regular expression synthesizer. Next, we analyse the time performance of several techniques included in FOREST. Finally, we close up in Chapter 7, where we summarise the main conclusions of this work, and discuss future directions within this topic.

Chapter 2

Background

This chapter offers an introduction to some key notions required to understand the rest of this document. We start off by looking into formal languages and associated definitions in Section 2.1, where we also define regular expressions and capturing groups, the main goals of our synthesis procedure[1]. In Section 2.2 we introduce some well-known logic problems as well as the necessary definitions and notation[2]. Finally, in Section 2.3, we present an overview of the basic concepts of program synthesis.

2.1 Regular Languages

Formal languages differ from the common meaning of the word *language* in that they are built from a set of well-defined rules and thus stringently specified. Programming languages, such as `C` or `Python`, are examples of formal languages. They have a strict syntax and no deviation is tolerated: if a string does not comply with the syntax rules, it is not in the language.

To define a formal language, we must first define an alphabet. An alphabet is a nonempty finite set whose elements are called the symbols, letters or tokens. Symbols of the alphabet can be, for example, letters, digits, or punctuation marks. Alphabets are usually identified by capital Greek letters.

Definition 2.1.1 (Alphabet)**.** An alphabet Σ is a nonempty finite set of symbols.

Example 2.1.1. The following are examples of alphabets:

- $\Sigma_1 = \{0, 1\}$: the binary alphabet,
- $\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$: the Latin lowercase alphabet.

Typically, the alphabets used in programming applications are larger than Σ_1 or Σ_2. The ASCII alphabet, for instance, contains 128 symbols which include all non-accentuated upper- and lower-case

[1] Definitions and examples in Section 2.1 were, in part, adapted from Chapter 3 of *Compilers: Principles, Techniques, and Tools*, by Aho, Lam, Sethi, and Ullman [2] and Chapter 2 of *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, by Rozenberg and Salomaa [59]

[2] Definitions and examples in Section 2.2 were, in part, adapted from *Handbook of Satisfiability*, by Biere, Heule, van Maaren, and Walsh [7] and Miguel Neves's MSc. Thesis: *Distributed Solver for Maximum Satisfiability* [68]

letters, digits, and most punctuation marks. Unicode characters, which include 154 modern and historical scripts, as well as multiple symbol sets, are more inclusive, and are the most commonly used in practice. In its current version, Unicode 13.0, this alphabet contains close to 150 thousand characters[3].

Given an alphabet, we can define strings (also called sentences or words) over it, which result from the concatenation of a finite number of the alphabet's symbols. Strings are usually identified by lower case letters and written between single quotes. The length of a string s, denoted $|s|$, is the number of symbols in it.

Definition 2.1.2 (String)**.** A string s over an alphabet Σ is a finite sequence of symbols drawn from Σ.

Definition 2.1.3 (Length of a string)**.** The length of a string s, $|s|$, is the number of symbols in s.

Example 2.1.2. The following are examples of strings:

- ϵ, the empty string, is the string of length zero,
- $s_1 =$ *synthesis* is a string over alphabet Σ_2 and it has length $|s_1| = 9$.

At last, a language is a (finite or infinite) countable set of strings over some alphabet. Languages are usually identified by calligraphic capital letters.

Definition 2.1.4 (Language)**.** A language $\mathcal{L}$ is a countable set of strings over some fixed alphabet Σ.

Example 2.1.3. The following are examples of languages:

- $\emptyset$, the empty set, is a language which contains no strings,
- $\{\epsilon\}$ is the language that contains only the empty string.

2.1.1 Regular Operations

Regular operations are a type of operation over languages, and regular languages are a subset of languages that are closed under regular operations. The three most basic regular operations are union, concatenation and closure.

The union of two languages $\mathcal{L}$ and $\mathcal{M}$, denoted $\mathcal{L} \cup \mathcal{M}$, is the set of all strings in $\mathcal{L}$ and all strings in $\mathcal{M}$ (including those that are in both). The concatenation of two languages $\mathcal{L}$ and $\mathcal{M}$, denoted $\mathcal{L}\mathcal{M}$, is the set all strings formed by taking any string from $\mathcal{L}$ and any string from $\mathcal{M}$ and concatenating them. The concatenation of the same language, $\mathcal{L}$, n times is sometimes denoted as $\mathcal{L}^n$. For example $\mathcal{L}^3 = \mathcal{L}\mathcal{L}\mathcal{L}$. The Kleene closure[4] (also called Kleene star or simply star) of a language $\mathcal{L}$, denoted $\mathcal{L}^*$, is the set of strings resulting from the concatenation of any string in $\mathcal{L}$ zero or more times. The Kleene closure of any language always includes the empty string.

Definition 2.1.5 (Regular operations)**.** There are 3 regular operations on languages, union, concatenation and closure, which are defined as follows:

[3] `https://www.unicode.org/charts`. Accessed on 24th October, 2020

[4] Kleene closure owes its name to the American mathematician Stephen Cole Kleene, who first described the concepts of regular language and regular expression in the 1950s.

Name	Operation on regular expressions	Operation on languages
Union	$r\|q$	$\mathcal{L}(r) \cup \mathcal{L}(q)$
Concatenation	rq	$\mathcal{L}(r)\mathcal{L}(q)$
Kleene closure	r^*	$\mathcal{L}(r)^*$

Table 2.1: Regular operations

- Union: $\mathcal{L} \cup \mathcal{M} = \{s : s \in \mathcal{L} \text{ or } s \in \mathcal{M}\}$,
- Concatenation: $\mathcal{L}\mathcal{M} = \{st : s \in \mathcal{L} \text{ and } t \in \mathcal{M}\}$,
- Kleene closure: $\mathcal{L}^*$ = $\bigcup_{i=0}^{\infty} \mathcal{L}^i$.

2.1.2 Regular Expressions

Regular expressions (often shortened to *regexes*) are used to describe regular languages. The language defined by a regular expression r is represented as $\mathcal{L}(r)$. Regular expressions are built recursively out of smaller regular expressions connected by operators. The simplest regular expressions refer to languages that contain just one symbol and are represented as that symbol in monospaced font.

Example 2.1.4. The regular expression `a` defines the language $\mathcal{L}(\texttt{a}) = \{a\}$.

All regular operations have a counterpart operation on regular expressions. For example, the union of two regular expressions r and q results in a regular expression that defines the regular language $\mathcal{L}(r) \cup \mathcal{L}(q)$. Since regular languages are closed under regular operations, it is possible to apply these operations to any regular expression. The operations in regular expressions are generally represented in the same way as their regular-language counterparts, with the exception of union, which is represented using a | instead of the usual set notation $\cup$. The operations on regular expressions and their equivalent regular operations are defined in Table 2.1.

The unary operator * has highest precedence, concatenation has second highest precedence and operator | has lowest precedence. All three operators are left associative. Parentheses can be used to ensure a sub-expression takes precedence over the rest.

Example 2.1.5. `a|b*c` is a regular expression that defines the language

$$\mathcal{L}(\texttt{a|b*c}) = \{a\} \cup \{b\}^*\{c\}$$

over the alphabet $\Sigma = \{a, b, c\}$. This language contains strings that are either a single a or zero or more bs followed by one c.

If two regular expressions r and s denote the same regular language, we say they are equivalent and write $r = s$. For instance, `a|b` = `b|a`.

Since the introduction of regular expressions with the basic operators for union, concatenation, and Kleene closure, some new operators have been defined for regular expressions with the purpose of eas-

ing the specification of certain string patterns. Here we show some of those commonly used operators: $+$, $?$, $\{m\}$, $\{m,n\}$, and character classes.

The unary postfix operator $+$ represents positive closure, which can be interpreted as 'one or more occurrences of' (note the similarity to Kleene closure, 'zero or more instances of'). It can be defined as

$$r^+ = rr^* = r^*r,$$

and gives rise to a new algebraic law:

$$r^* = r^+|\epsilon.$$

The unary postfix operator $?$, often called *option*, means 'zero or one occurrences of', and can be defined as

$$r? = r|\epsilon.$$

The range operators, $\{m\}$ and $\{m,n\}$, specify how many occurrences of the preceding regular expression they match. $\{m\}$ is interpreted as 'exactly m copies of', and $\{m,n\}$ is interpreted as 'at least m and at most n copies of'. For example:

$$r\{3\} = rrr,$$

$$r\{1,3\} = r|rr|rrr = rr?r?.$$

The operators $+$, $?$, $\{m\}$ and $\{m,n\}$ have the same precedence and associativity as operator $*$.

Character classes are a form of regular expression shorthand notation. A regular expression $a_1|a_2|a_3|...|a_n$, where each a_i is a symbol of the alphabet, can be replaced by the shorthand $[a_1a_2...a_n]$. When $a_1a_2...a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last tokens separated by a hyphen.

Example 2.1.6. The following are examples of character classes:

- [abc] = a|b|c
- [a-z] = a|b|...|z
- [a-z0-9] = a|b|...|z|0|1|...|9

2.1.3 Capturing Groups

Regular expressions can be coupled with capturing groups. A capturing group is a sub-expression within a regular expression, usually specified with parentheses. When a string is matched by a regular expression with capturing groups, the match operation returns the resulting captures, i.e., the text that is matched by the regular expression inside each pair of parentheses. Capturing groups are used to extract information from text. Once extracted, that information can be used independently from the original string. Besides producing a capturing group, the parentheses also cause the expression inside them to take precedence over the remaining operators in the regular expression. For instance, if a quantifier is placed after the parentheses, it applies to the regular expression inside as a whole.

Example 2.1.7. Consider the regular expression r_1 = `[0-9]{2}/[0-9]{2}/[0-9]{4}` and the string s = "19/08/1996". The regex r_1 matches the string s. Because r_1 has no capturing groups, no captures are produced. Consider now r_2 = `([0-9]{2})/([0-9]{2})/([0-9]{4})`. r_2 also matches s (it has the same base regular expression as r_1) and it has 3 capturing groups. Thus, when matched to s, r_2 produces the captures "19", "08", and "1996".

In the domain of form validations, we focus primarily on the capture of integer values in the input strings. We use the notation $\$i, i \in \{0, 1, ...\}$, to refer to the integer value of the text captured by the $(i+1)^{\text{th}}$ group.

Example 2.1.8. Recall the string s = "19/08/1996" and regex r_2 = `([0-9]{2})/([0-9]{2})/([0-9]{4})` from Example 2.1.7. When matched to s, r_2 produces the captures "19", "08", and "1996". Thus, $\$0 = 19$, $\$1 = 08$, and $\$2 = 1996$.

2.2 Constraint Solving

Constraint solving techniques are used to solve problems where the space of solutions is restricted by logical constraints. Throughout this document we make extensive use of Satisfiability Modulo Theories (SMT) and Maximum Satisfiability Modulo Theories (MaxSMT) to model problems in our synthesis domain. The SMT problem can be seen as a generalisation of the Propositional Satisfiability (SAT) problem. Similarly, the MaxSMT problem is defined as a generalisation of Maximum Satisfiability (MaxSAT). The next sections provide a brief introduction to each of these formalisms.

2.2.1 Propositional Satisfiability

In logic and computer science, SAT is the problem of, given a Boolean formula, determining if there exists an assignment of its variables that satisfies it. In addition to its theoretical importance, SAT has many practical applications in the field of Computer Science. In 1971, SAT was the first problem proven to be NP-Complete [11]. This means that numerous decision problems can be reduced to the SAT problem, and subsequently solved using an off-the-shelf SAT solver.

Definition 2.2.1 (Literal)**.** A literal l is a Boolean variable ($l = x$) or its complement ($l = \neg x$).

Definition 2.2.2 (Clause)**.** A clause c is a disjunction of literals: $c = l_1 \vee l_2 \vee ... \vee l_k$.

SAT formulas are usually represented in the Conjunctive Normal Form (CNF).

Definition 2.2.3 (CNF Formula)**.** A formula ϕ in CNF is a conjunction of clauses: $\phi = c_1 \wedge c_2 \wedge ... \wedge c_n$.

Example 2.2.1. Consider the variables $X = \{x_1, x_2, x_3\}$. Then, $\phi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$ is a CNF formula over X.

Definition 2.2.4 (Assignment)**.** Given a formula ϕ, an assignment is a mapping $\nu : X \rightarrow \{\textit{True}, \textit{False}\}$, where X is the set of variables in ϕ.

Given an assignment $\nu : X \to \{\textit{True}, \textit{False}\}$ and a variable $x \in X$, the positive literal x is satisfied by ν if and only if $\nu(x) = \textit{True}$ and the negative literal $\neg x$ is satisfied by ν if and only if $\nu(x) = \textit{False}$; a clause is satisfied by ν if and only if at least one of its literals is satisfied by ν; a formula in CNF is satisfied by ν if and only if all of its clauses are satisfied by ν.

Definition 2.2.5 (Model)**.** Given a propositional formula ϕ and an assignment ν, ν is a model of ϕ if and only if ν satisfies ϕ.

Example 2.2.2. Recall the CNF formula from Example 2.2.1: $\phi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$. The assignment $\nu_1 = \{x_1 \mapsto \textit{False}, x_2 \mapsto \textit{True}, x_3 \mapsto \textit{True}\}$ is a possible model of ϕ_1.

2.2.2 Maximum Satisfiability

The problem of finding an assignment ν to the variables of a CNF formula that satisfies the maximum number of clauses possible is known as MaxSAT. Unlike SAT, which is a decision problem, MaxSAT is an optimisation problem. In computer science, many optimisation programs can be reduced into MaxSAT. Moreover, MaxSAT can be used used to get insights about the *unsatisfiable* instances. A SAT solver tells us that we cannot satisfy all clauses. A MaxSAT solver tells us how many can be satisfied at most.

Example 2.2.3. Consider the CNF formula $\phi_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge \neg x_1$. No assignment can satisfy all clauses in this formula. However, there can be assignments that satisfy two of the clauses in ϕ_2. For example, $\nu_2 = \{x_1 \mapsto \textit{True}, x_2 \mapsto \textit{True}\}$ satisfies all but the last clause in ϕ_2: it is a MaxSAT model for ϕ_2.

An important variation of the MaxSAT problem, known as *partial* MaxSAT, divides its clauses into two types: hard clauses, ϕ_H, and soft clauses, ϕ_S. Partial MaxSAT is then the problem of finding an assignment ν to the variables of a CNF formula that satisfies all hard clauses and as many soft clauses as possible.

Definition 2.2.6 (Model)**.** Let (ϕ_H, ϕ_S) be an instance of partial MaxSAT. An assignment ν is a model for (ϕ_H, ϕ_S) if and only if ν satisfies all clauses in ϕ_H and the maximum number possible in ϕ_S.

Example 2.2.4. Consider an instance of partial MaxSAT (ϕ_H, ϕ_S), with the hard clause $\phi_H = \neg x_1$ and soft clauses $\phi_S = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge \neg x_2$. The assignment $\nu_2 = \{x_1 \mapsto \textit{False}, x_2 \mapsto \textit{True}\}$ is a model for (ϕ_H, ϕ_S), because it satisfies all hard clauses and as many soft clauses as possible.

2.2.3 Satisfiability Modulo Theories

Oftentimes, it is advantageous to express the problem at hand using more expressive logics. In this thesis we make extensive use of an extension of SAT: Satisfiability Modulo Theories (SMT). SMT is the problem of satisfiability of formulas with respect to some background theory $\mathcal{T}$, which defines the interpretations of certain function symbols. This allows us to incorporate fragments of first-order logic in CNF formulas.

Definition 2.2.7 ($\mathcal{T}$-atom)**.** A $\mathcal{T}$-atom t is a ground atomic formula in $\mathcal{T}$.

Figure 2.1: Program Synthesis

Definition 2.2.8 ($\mathcal{T}$-literal)**.** A $\mathcal{T}$-literal is a $\mathcal{T}$-atom (t) or its negation ($\neg t$).

Definition 2.2.9 ($\mathcal{T}$-formula)**.** A $\mathcal{T}$-formula is then analogous to a propositional formula but it is composed of $\mathcal{T}$-literals instead of propositional literals.

The values in an SMT assignment depend on the background theory $\mathcal{T}$. Commonly used theories include the theory of Linear Integer Arithmetic (LIA), whose values are integer, the theory of *strings*, whose values are strings, and the theory of Equality with Uninterpreted Functions (EUF), which assigns functions with a valid interpretation. Given an SMT assignment, ν, and a $\mathcal{T}$-atom t, the positive $\mathcal{T}$-literal t is satisfied by ν if and only if $\nu(t)$ is assigned *True* according to theory $\mathcal{T}$ and the negative $\mathcal{T}$-literal $\neg t$ is satisfied by ν if and only if $\nu(t)$ is assigned *False* according to $\mathcal{T}$; a $\mathcal{T}$-formula is satisfied by ν if and only if all of its clauses are satisfied by ν.

SMT is then the problem of deciding, given an SMT formula ϕ, if there exists an assignment of the variables and functions of ϕ that satisfies it.

Example 2.2.5. Let $\phi_3 = (b - a = 1) \wedge ((b < 5) \vee (a > 10))$ be a SMT formula in the theory of LIA. A possible model for ϕ_3 is $\nu_3 = \{a \mapsto 1, b \mapsto 2\}$.

2.2.4 Maximum Satisfiability Modulo Theories

The same way MaxSAT finds the maximum number of clauses that can be satisfied in a CNF formula, MaxSMT finds the maximum number of clauses that can be satisfied in an unsatisfiable $\mathcal{T}$-formula. We can also define *partial* MaxSMT with hard clauses, ϕ_H, and soft clauses, ϕ_S.

Example 2.2.6. Let (ϕ_H, ϕ_S) be a partial MaxSMT instance, with hard clauses $\phi_H = (b-a = 1) \wedge (b < 10)$ and soft clauses $\phi_S = (b < 5) \wedge (a > 10)$. We can satisfy at most one of the soft clauses. Therefore $\nu_3 = \{a \mapsto 1, b \mapsto 2\}$ is an optimal model for this instance.

2.3 Program Synthesis

Program synthesis is the task of automatically generating a program that satisfies some desired behaviour expressed as a high-level specification (see Figure 2.1). This specification can range from a complete formal definition, such as a first-order formula [20, 24], to more ambiguous descriptions of the desired program's behaviour, like a set of input-output examples [5, 17, 28, 38, 42, 56, 60, 67] or a natural language sentence [8, 13, 27, 38, 56, 73].

According to Gulwani et al. [21, 25], a program synthesizer is typically characterised by three key dimensions: (i) the way the user specifies the desired characteristics of the program, (ii) the space of all possible programs the synthesizer can generate, and (iii) the search technique used to explore that space.

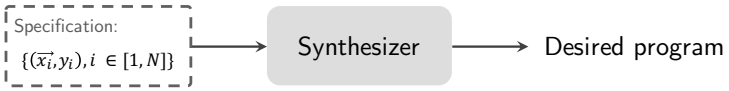


Figure 2.2: PBE: The desired behaviour specification is a set of N input-output examples.

Desired behaviour specification is the most important characteristic of a synthesizer from the users' point of view. It is the language in which they describe the behaviour of the program they intend to generate. It must take into consideration the underlying task, the users' technical background and which material is available when using the synthesizer.

Program space is the set of all programs the synthesizer can possibly generate. In other words, it is the space over which the synthesizer searches for a feasible program. It depends on the domain of the problem the synthesizer is intended to solve. It must be expressive enough to ensure the desired program is included, but also restricted enough lest the search problem become intractable.

Search technique refers to the method employed to find the desired program within the program space. It can be based on enumerative search, deductive search, constraint solving, or some combination thereof.

The three dimensions are further described in sections 2.3.1, 2.3.2 and 2.3.3, respectively.

2.3.1 Desired Behaviour Specification

For the synthesis procedure to start, the user must first specify the program's intended behaviour. The desired behaviour can be described in many different ways, and is internally converted to some sort of behavioural constraints, which the output program must satisfy.

Definition 2.3.1 (Desired Behaviour Specification)**.** The desired behaviour specification is a predicate ϕ, such that $\phi(\vec{x}, y)$ is *True* if and only if y is the desired output value for the input vector $\vec{x}$.

The first approaches to automating the creation of programs, proposed in 1969 [20, 69], were based in deductive synthesis. Such synthesizers work based on a complete formal specification of the desired program's behaviour. Then, they employ a theorem prover to construct a proof of the provided specification from which it is able to extract the executable program [20, 36, 37, 69].

Deductive synthesizers require the user to provide a complete formal description of the desired behaviour, such as a first-order formula. Because it is a complete definition, the synthesizer always returns a program with the exact desired behaviour, and it is always satisfactory to the user. However, writing these kind of specifications can be as complex as writing the program itself, and might force the user to learn a new formalism in any case.

Such difficulties motivated a new approach to program synthesis: inductive synthesis [15, 17, 32, 44, 45, 47, 53, 54, 60, 67]. The program is then built based on simpler (albeit ambiguous) specifications, easier for the user to devise. Programming by Example (PBE) is a branch of inductive synthesis

where the user intent is specified using input-output examples [18, 19, 42, 60, 67, 70, 71], as shown in Figure 2.2.

Definition 2.3.2 (Input-Output Examples)**.** Input-output examples are a type of incomplete specification defined as a set of N tuples: $\mathcal{X} = \{(\vec{x_i}, y_i), i \in \{1, ..., N\}\}$ where each y_i is the desired return value for input $\vec{x_i}$.

Definition 2.3.3 (Programming by Example)**.** Programming by example is the problem of synthesising a program using input-output examples as a specification of the desired program behaviour. In programming by example, the behavioural constraint is

$$\bigwedge_{(x_i, y_i) \in \mathcal{X}} P(x_i) = y_i,$$

which states that program P is correct if it yields the correct output for the inputs specified in the specification.

Input-output examples, although easier for the users to obtain and comprehend, are an incomplete specification. In general, there are several programs consistent with the provided specification even though not all of those correspond to the desired program and might not exhibit the behaviour expected by the user for cases that are not covered by the specification.

The ambiguity of input-output examples raises the necessity of selecting one among multiple candidate programs. One way to do this selection is by ranking the correct programs according to the measure of some characteristic that is desirable in programs of the domain in question, and returning to the user the program that ranks the highest [8, 14, 26, 51, 54, 61]. Common desirable characteristics used to rank the programs include: execution speed (when we are looking for an efficient program), robustness (how well it generalises to new input-output examples), and readability (favours common operations in the underlying language, making it easy for the user to understand). Another approach is to enable the synthesizer to interact with the user to try and disambiguate the underlying intent [23, 33, 40, 54, 70, 71]. Section 3.4 provides more details on this topic.

2.3.2 Program Space

The next step towards finding a program that satisfies the user's needs is defining the space of programs over which the synthesizer performs the search: the program space. We cannot consider all the programs that can be written using a full-featured programming language. Too many programs would be taken into consideration, rendering the search space intractable. On that account, we need to restrict the language in which the programs are written in order to enable an efficient search of the program space. However, we must ensure it remains expressive enough to capture many real-world tasks within the considered specialised domain. We call this language Domain Specific Language (DSL). The choice of a synthesizer's DSL is crucial: it must allow a good balance between expressiveness and efficiency.

Definition 2.3.4 (Domain Specific Language)**.** Domain specific language is the restricted language in

which the synthesised programs are written. It includes information about both form (syntax) and meaning (semantics).

We may impose restrictions on the allowed datatypes, as well as the operations over them so as to include only those relevant for the considered domain. For example, one could allow only integers and comparison operations, or arithmetic operations, or even only operations supported in some API exported by a given library. We can also restrict the program space by imposing constraints over the control structure of the program: we may disallow looping structures in the program, or bound the number of statements.

The constraints imposed on the language are named structural constraints, and they define the search space the synthesiser must consider. A CFG is typically used to define the syntax of the DSL.

Definition 2.3.5 (Context-Free Grammar[5])**.** A CFG is defined by a 4-tuple (N, Σ, R, S), where:

- N is a set of non-terminal symbols,
- Σ is a set of terminal symbols (disjoint from N),
- R is a set of rules or productions, each of the form $A \to \beta$, where:
  - A is a non-terminal,
  - β is a string of symbols from the infinite set of strings $(\Sigma \cup N)*$, and
- S is a designated start symbol and a member of N.

Then, a program on a DSL is a production of the corresponding CFG containing only terminal symbols, which include all operators and literal values in the DSL.

Syntax-guided synthesis [4, 42] is the branch of program synthesis in which the user to supplies the language syntax (in the form of a grammar) alongside the behaviour specification. This provides structure to the program space, which may allow for a more efficient search method. Furthermore, the generated programs are more interpretable to the user and better adapted to the domain at hand, since they are derived from the given grammar. On the other hand, syntax-guided synthesis requires the user to have a deeper technical knowledge not only on the specific domain on which he or she is working but also on formal languages and how to define them.

2.3.3 Search Technique

Program synthesis can be seen as a search problem. It aims at finding a program in the search space defined by structural constraints that satisfies the behavioural constraints.

Definition 2.3.6 (Program Synthesis)**.** Given a specification of desired behaviour ϕ, program synthesis is the problem of finding a program P that satisfies ϕ.

Several search techniques have been explored to solve program synthesis. In the remainder of this section, some of these search techniques are briefly explained. Note that one synthesizer does not have to apply only one of these techniques; more often, a combination of several techniques is applied.

[5] Adapted from Chapter 2 of *Speech and Language Processing* by Jurafsky and Martin [29].

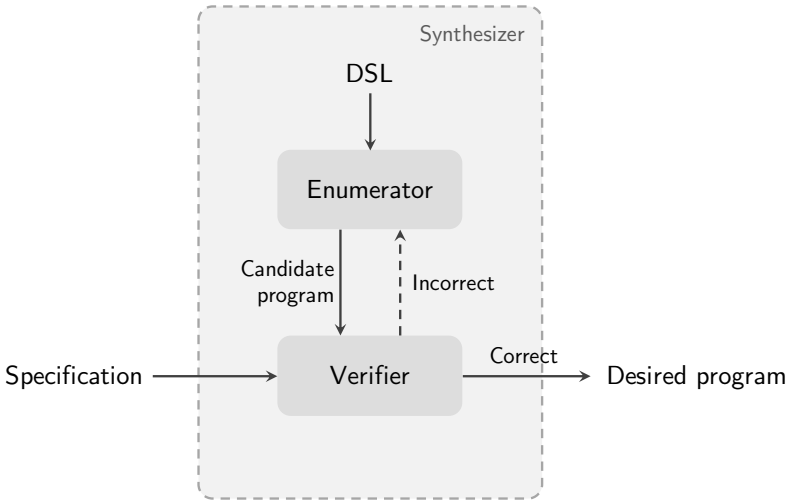


Figure 2.3: Enumerative search

Enumerative Search [15, 17, 32, 45, 47, 58] is a common approach to solve the search problem of program synthesis. In this technique, the synthesizer must include two key components: an *enumerator* and a *verifier*. The *enumerator* successively enumerates programs from the program space in some order. For each candidate program, the *verifier* subsequently checks whether it satisfies all behavioural constraints, i.e., it is consistent with the user intent specification. If the *verifier* deems the program consistent with the given specification, then the program is correct and it can be returned to the user. If, on the other hand, the *verifier* decides that the given program does not satisfy the specification, then the *enumerator* must pick a new program from the search space. This loop is described in Figure 2.3.

A naive implementation of enumerative search does not scale to complex programs; however, it is a very effective strategy when coupled with some optimisation techniques. First off, the program space must be structured according to some metrics, usually program size or complexity, such that the *enumerator* yields simpler programs first and only when those are refuted by the *verifier* are more complex programs considered. This concept contributes to solving the problem of user intent ambiguity introduced in Section 2.3.1 by applying the Ockham's Razor principle: choose the simplest program that is consistent with the specification.

Another way to speed up enumerative search is by employing pruning techniques, such as discarding equivalent programs from the search space. In addition, when the *verifier* rejects a program, it may apprise the *enumerator* of the reason behind the program's incorrectness, thus reducing the program space and preventing the *enumerator* from generating programs that fail in the same way. This idea is further discussed in Section 3.2.

Deductive Search [10, 35, 51] is based on a top-down propagation of constraints through the grammar that defines the program space. It recursively reduces the problem of synthesising an expression that satisfies a certain specification to simpler sub-problems, thereupon combining those results.

Suppose we want to synthesise an expression e of the form $F(e_1, e_2)$ that complies with a specification ϕ. Deductive search would leverage the inverse semantics of F to push constraints on e down

through the grammar into constraints on e_1 and e_2. Finding e_1 and e_2 are then simpler sub-problems whose solutions, once found, can be combined to produce the originally desired expression e.

Deductive search is very convenient when the underlying grammar allows for a rich set of constants. In such cases, enumerative search is no longer viable: an enumeration step is required for each possible constant in order to find the right one, resulting in too many iterations. On the other hand, the top-down deductive technique can deduce constants based on the accumulated constraints as the last step in the search process.

Constraint Solving [62, 66] consists on somehow generating a logical formula whose solution yields the intended program, and then solving it.

Several approaches can be used to generate the formula. On one extreme, we have invariant-based methods, which generate one formula that is satisfiable if and only if the program is consistent with the specification. Upon solving such a formula, we have not only a correct program but also a inductive proof of its correctness. However, these methods do not scale well with the complexity of the specification — the resulting formula may be intractable, and much more complex than the program itself. To circumvent this drawback, we can instead make use of input-based methods. Then, the formula asserts the correctness of the program only on a subset of all possible inputs, which leads to simpler constraints.

Once we have a logic formula, it must be solved. These formulas are of second-order logic, and need to be first reduced to first-order quantifier-free constraints that can the be solved using an off-the-shelf logic-based solver. Second-order reduction can be achieved using techniques such as CEGIS [1, 62], which is described in Section 3.2.

Chapter 3

Program Synthesis

Program synthesis is a very wide field of research. It has been studied by several different research communities and applied to diverse problems. As such, many approaches have been proposed to deal with the specific challenges introduced by each application, resulting in a great variety of focused algorithms that aim at finding a better program in less time. In this chapter, we take a more in-depth look into some of these techniques.

In Section 3.1 we introduce sketch-based enumeration: an enumeration method that deals with partial programs. In Section 3.2, we look at a method that steers the search in the right direction by using information about wrong answers to avoid future similar mistakes. In Sections 3.3 and 3.4, we look at ways to deal with the ambiguity of the desired behaviour specification in inductive synthesis. Finally, in Section 3.5, we take an in-depth look at two existing regular expression synthesizers: ALPHAREGEX and REGEL.

3.1 Sketch-based Enumeration

Some synthesizers perform enumerative search using partial programs [8, 15, 16, 53, 62–65, 70, 73]. Instead of completely defining a program, partial (or incomplete) programs contain holes alongside the DSL components. For a partial program to become a syntactically correct complete program, all its holes must be filled with syntactically consistent expressions. If we think in terms of the DSL's representation as a CFG, a *complete* program is a production of the grammar containing only terminal symbols, i.e, only operators and literal values in the DSL. In contrast, a *partial* program is a production of the grammar that may contain non-terminal symbols. Since each non-terminal symbol can correspond to many terminal symbols, a partial program can represent many complete programs.

Sketch-based enumeration deals with a particular type of partial programs: *sketches*. A sketch is a partial program where the holes cannot be filled with operators: all missing constructs are literal values of the DSL. Instead of enumerating complete programs, a sketch-based synthesizer takes one of two approaches: Either (i) a sketch is provided by the user, who already possesses a high-level description of the desired program, in which case the synthesizer must complete it according to the specification,

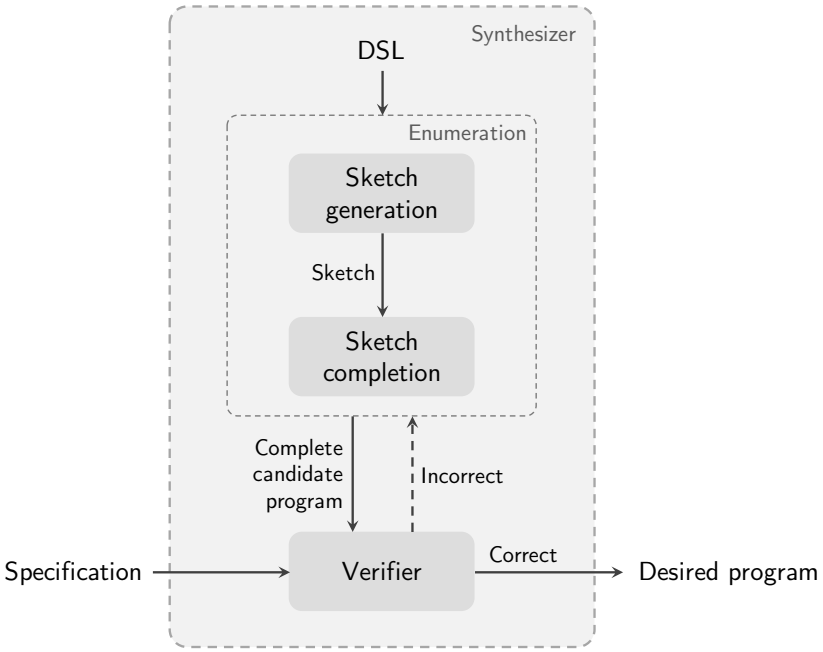


Figure 3.1: Sketch-based enumeration

or (ii) the synthesizer is responsible for both producing a suitable sketch and completing it. The enumeration process is then split in two steps: sketch generation and sketch completion. During sketch generation, the synthesizer enumerates incomplete programs in some order. During sketch completion, the synthesizer enumerates complete programs for each given sketch, by successively filling each hole with a syntactically correct expression. Sketch-based enumeration is outlined in Figure 3.1.

In many synthesizers, two-step enumeration outperforms normal enumeration because there is a very efficient way to fill the sketch holes, either because the few possible values exist for each hole, or because they can be directly deduced from the specification (instead of enumerated).

3.2 Counterexample Guided Inductive Synthesis

Program synthesis is a search problem where the goal is to find a program P that satisfies a given specification ϕ. $\phi(\vec{x}, y)$ is *True* if and only if y is the desired output value for input $\vec{x}$. We can define the program synthesis problem with the following logic formula:

$$\exists P \; \forall \vec{x}, y : (P(\vec{x}) = y) \Rightarrow \phi(\vec{x}, y). \tag{3.1}$$

Due to the existential quantifier over function P, (3.1) is a second-order formula and, as such, it is generally undecidable. To work around this problem, we may note that even though *finding* a program that satisfies a specification may be infeasible, *verifying* that a given program P satisfies a specification is a first-order problem:

$$\forall \vec{x}, y : (P(\vec{x}) = y) \Rightarrow \phi(\vec{x}, y). \tag{3.2}$$

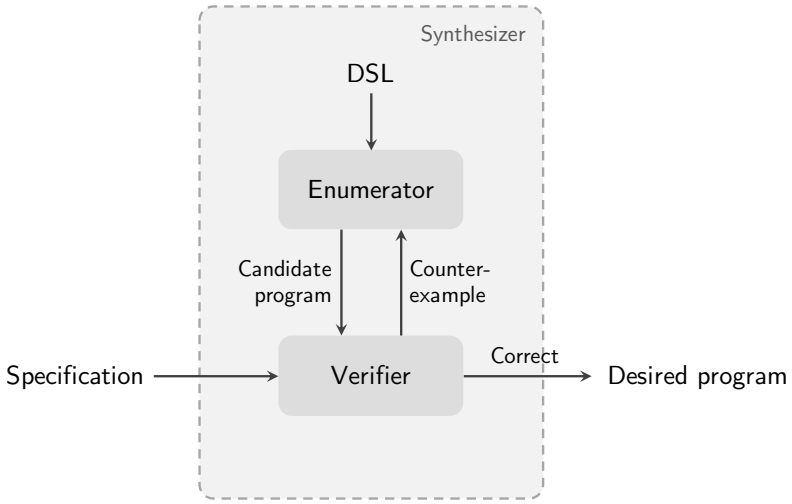


Figure 3.2: CEGIS

Formula (3.2) is a first-order formula and can be solved using an off-the-shelf first-order solver. Any program P that satisfies (3.2) is a correct program, i.e., it complies with the specification ϕ. Instead of proving (3.2), we can equivalently disprove its negation:

$$\exists \vec{x}, y : (P(\vec{x}) = y) \wedge \neg\phi(\vec{x}, y). \tag{3.3}$$

Formula (3.3) is also a first-order formula. Formula (3.3) is unsatisfiable if and only if P is a correct program. Therefore, to find a correct program, we search the program space for a program P such that (3.3) is unsatisfiable. Once found, P can be returned to the user.

Whenever we encounter a program P for which formula (3.3) is satisfiable, the values of $\vec{x}$ and y that satisfy the formula constitute a counterexample: an input $\vec{x}$ for which P does not satisfy the specification ϕ; in other words, an input $\vec{x}$ to which our program returns the wrong output. If we take this counterexample $(\vec{x}, y)$ into account during our subsequent search, none of the new candidate programs returns y on the input $\vec{x}$. We have thus strengthened the specification, eliminating the previous incorrect candidate program.

Definition 3.2.1 (Counterexample)**.** A counterexample $(\vec{x}, y)$ for an incorrect program P is an input-output pair such that $P(x) = y$ and $\neg\phi(\vec{x}, y)$, i.e., y is the output returned by P on input $\vec{x}$ even though $(\vec{x}, y)$ is not consistent with the specification.

This approach was proposed in 2008 by Solar-Lezama in his PhD Thesis [62] and it is represented in Figure 3.2. In this figure, the *verifier* is a solver that tries to satisfy formula (3.3).

We can further improve this method by reformulating the verification formula in order to produce a constructive counterexample:

$$\exists \vec{x}, y : (P(\vec{x}) \neq y) \wedge \phi(\vec{x}, y). \tag{3.4}$$

Like before, (3.4) is a first-order formula and it is unsatisfiable if and only if P is a correct program. When (3.4) is satisfiable P is not a correct program and the values of $\vec{x}$ and y that satisfy the formula are now a *constructive* counterexample: y is the correct output for input $\vec{x}$. While a model $(\vec{x}, y)$ that satisfies (3.3)

is an input-output pair that the program must *not* satisfy, a model $(\vec{x}, y)$ that satisfies (3.4) is a correct input-output pair that the program *must* satisfy. Adding the constructive counterexample to our previous specification results in a much stronger constraint and prunes the remaining search space even further.

Definition 3.2.2 (Constructive counterexample)**.** A constructive counterexample $(\vec{x}, y)$ for an incorrect program P is an input-output pair such that $P(x) \neq y$ and $\phi(\vec{x}, y)$, i.e., y is not the output returned by P on input $\vec{x}$ even though $(\vec{x}, y)$ is consistent with the specification.

3.3 Oracle Guided Inductive Synthesis

As discussed in Section 2.3.1, PBE comes with the drawback of incompleteness of the behaviour specification. Many programs are consistent with the provided specification, but not all these programs exhibit the user's intended behaviour.

In order to restore soundness of the solution, Jha et al. proposed in 2010 a new approach which makes use of distinguishing inputs to disambiguate the input-output examples: Oracle Guided Inductive Synthesis (OGIS) [28]. An *I/O oracle* maps any given input to the desired output and it is used as an alternative to a complete specification. This means that whenever the *I/O oracle* is queried on any input vector $\vec{x}$, it always returns the correct output y.

We start with the same schema we had for CEGIS (Figure 3.2), with a *verifier* which, upon receiving a program P, decides whether it is consistent with the behavioural constraints ϕ. However, in OGIS we no longer return to the user the first correct program we find.

When a correct program P_1 is found, it is stored, and the search continues for another correct program. If no other correct program is found, then P_1 is the unique correct program and it can be returned to the user. If another correct program P_2 is found, then we have two programs consistent with the provided specification, i.e., P_1 and P_2 satisfy the following formulas:

$$\forall \vec{x}, y : (P_1(\vec{x}) = y) \Rightarrow \phi(\vec{x}, y), \tag{3.5a}$$

$$\forall \vec{x}, y : (P_2(\vec{x}) = y) \Rightarrow \phi(\vec{x}, y). \tag{3.5b}$$

Upon finding two correct programs, we want to find an input to which the two programs P_1 and P_2 return different outputs: a *distinguishing input*. To produce a distinguishing input, we can try to solve:

$$\exists \vec{x} : P_1(\vec{x}) \neq P_2(\vec{x}). \tag{3.6}$$

If formula (3.6) is unsatisfiable then P_1 and P_2 are equivalent programs. In this case, either there are more correct programs that can be taken into consideration, or $P_1 = P_2$ is the unique program that satisfies all input-output examples and it is returned to the user. If (3.6) is satisfiable, the distinguishing input $\vec{x}$ can be extracted from the model $(\vec{x}, y_1, y_2)$ that satisfies it. Once found, we can query the *I/O oracle* on the distinguishing input, who yields the correct output for it. The distinguishing input along with its correct output forms a new correct input-output pair $(\vec{x}, y)$ which can then be added to the specification ϕ, further

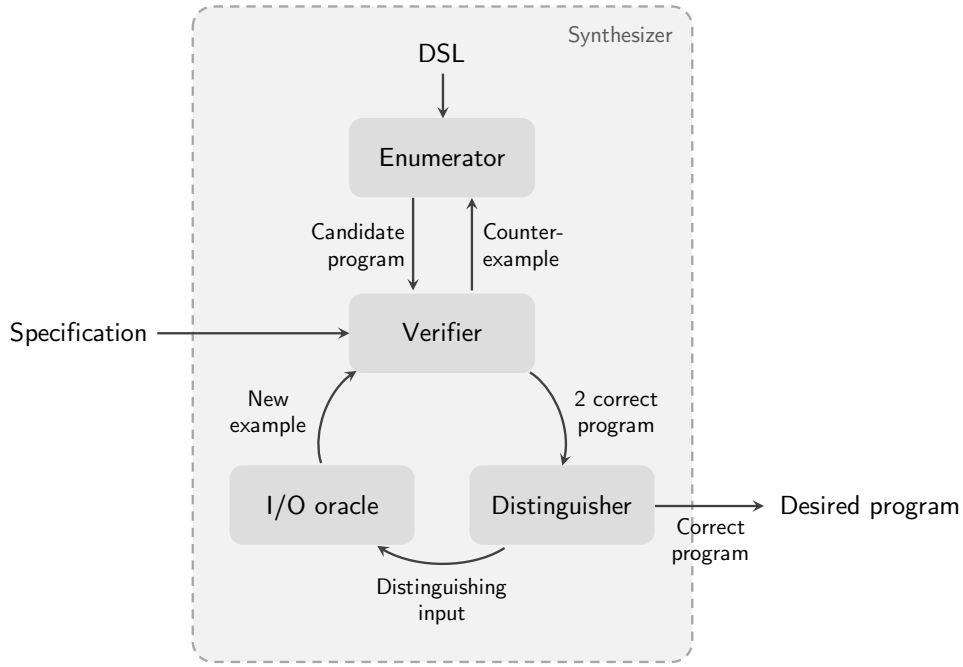


Figure 3.3: OGIS

constraining the search space.

In the end of this cycle, we have a set of input-output pairs that completely specify the generated program, thus eliminating all ambiguity.

This method is illustrated in Figure 3.3. In this figure, the *enumerator* and *verifier* are identical to those in CEGIS (Figure 3.2). The *distinguisher* is a new entity, which can be a solver that tries to satisfy formula (3.6).

3.4 User Interaction

As mentioned before, PBE uses input-output examples as the desired behaviour specification, which can be very ambiguous. When the synthesizer simply picks a correct program to return to the user, it may not satisfy the desired behaviour in corner cases not covered by the examples provided. In order to increase confidence in the synthesizer's solution, a good way to disambiguate the specification is by explicitly interacting with the user so he or she can provide further information about the intended program. In this section we look at a few different user interaction models that allow the synthesizer to gather more information about the intended program, thus resolving the ambiguity in the initially provided examples.

3.4.1 Conversational Clarification Model

Conversational clarification is an interaction method first described in 2015 by Mayer et al. [40] that has since been successfully integrated in many synthesizers [33, 43, 70, 71]. During the synthesis procedure, the synthesiser asks the user questions about certain inputs, and uses the answers to resolve ambiguities in the desired behaviour specification.

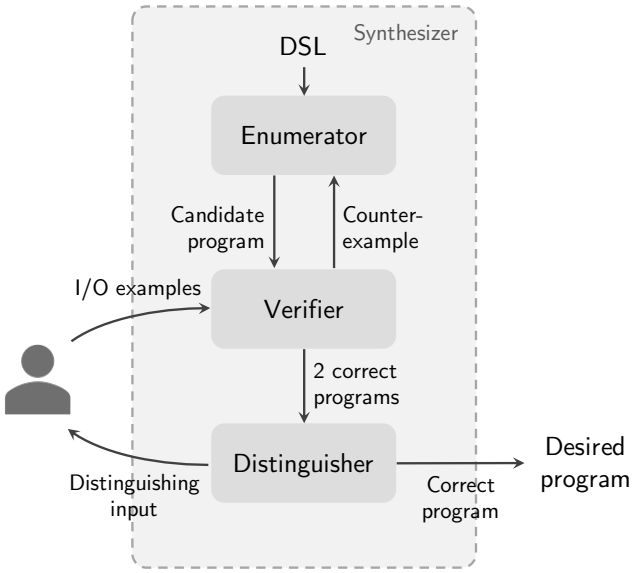


Figure 3.4: Conversational clarification

After the synthesizer has generated several programs that are consistent with the user-provided examples, it uses the *distinguisher* described in Section 3.3 to produce a distinguishing input: an input for which two correct programs yield two different outputs. The synthesizer then queries the user on what is the desired output for that specific input (it may be the output returned by one of the synthesised programs, or a different one altogether). The distinguishing input along with the desired output form a new input-output example, which is added to the specification, making it stronger. Each distinguishing input splits the search space in a different way, so the number of necessary user interactions depends on the chosen distinguishing input in each iteration. In the end, only one program remains. Since all ambiguity in the original specification has been resolved, the remaining program is the desired program and it can be returned to the user.

Conversational clarification is represented in Figure 3.4. It is no more than a variation of the OGIS method, described in Section 3.3, where the user plays the part of the *I/O oracle*.

3.4.2 OPTIONS Model

Ramos et al. [54, 55] propose two interaction models: OPTIONS and Y/N. Both models have in common with conversational clarification the fact that they interact with the user using distinguishing inputs, i.e., new inputs generated by the synthesizer whose output differs for different candidate programs. The main difference is that, while conversational clarification distinguishes only between two programs, OPTIONS and Y/N distinguish *optimally* between $n > 2$ candidate programs, $P_1, ..., P_n$. In each interaction they maximise the number of programs that are eliminated from the search space. Besides, unlike conversational clarification, with OPTIONS and Y/N, the program's output is always shown to the user beforehand. The user just has to select one among several or classify it as correct or incorrect. In sum, not only are OPTIONS and Y/N's interactions fewer, they are also easier for the user to answer.[1]

[1] UNCHARTIT's interaction models can be tried out at `http://sat.inesc-id.pt/unchartit/dist/`

The OPTIONS model shows the user a distinguishing input, as well as several options that correspond to the outputs of candidate programs for that input. Then, the user selects the correct output among the provided options. Ideally, there is a single input example that produces a different output for each candidate program. In this best-case scenario, a single interaction is sufficient to disambiguate all candidate programs with the OPTIONS model: only one is consistent with the user's selection.

To produce the distinguishing input, $\vec{x}$, we start by defining a Boolean variable b_{ij} for each pair $i, j \in \{1, ..., n\}, i < j$. b_{ij} is *True* if and only if P_i and P_j have the same output for $\vec{x}$:

$$\bigwedge_{i,j \in \{1,...,n\}, i<j} (P_i(\vec{x}) = P_j(\vec{x})) \leftrightarrow b_{ij}. \tag{3.7}$$

To ensure that at least two of the candidate programs produce a different output for the distinguishing input $\vec{x}$, we include the constraint:

$$\bigvee_{i,j \in \{1,...,n\}, i<j} \neg b_{ij}. \tag{3.8}$$

Finally, since it is our goal to find an input such that all programs have different outputs, we want as few b_{ij} set to *True* as possible. Thus, we add the following soft constraints:

$$\bigwedge_{i,j \in \{1,...,n\}, i<j} \neg b_{ij}. \tag{3.9}$$

We use a MaxSMT solver to try and find a model that satisfies the formula. If it is unsatisfiable, then there is no input $\vec{x}$ to which any of the programs have a different output, i.e., $P_1, ..., P_n$ are all equivalent. If it is satisfiable, then we show $\vec{x}$ to the user along with all possible outputs $P_i(\vec{x})$. The user selects the correct output, and the synthesizer discards all programs whose output for $\vec{x}$ differs.

3.4.3 Y/N Model

On the Y/N interaction model, the goal is to identify a distinguishing input $\vec{x}$ that splits the set of programs into two sets $\mathcal{A}$ and $\mathcal{B}$, such that all programs in $\mathcal{A}$ produce the same output o given input $\vec{x}$, and all programs in $\mathcal{B}$ produce outputs different than o given input $\vec{x}$. The goal is then to find $\vec{x}$ such that half the programs are placed in each set.

To encode this problem into MaxSMT, we have the same Boolean variables b_{ij} for all pairs $i, j \in \{1, ..., n\}, i < j$, which are *True* if and only if P_i and P_j have the same output for input $\vec{x}$:

$$\bigwedge_{i,j \in \{1,...,n\}, i<j} (P_i(\vec{x}) = P_j(\vec{x})) \leftrightarrow b_{ij}. \tag{3.10}$$

Then, we define a set of Boolean variables $p_i^{\mathcal{A}}$ (resp. $p_i^{\mathcal{B}}$) for all $i \in \{1, ..., n\}$ which are assigned *True* if and only if program P_i is placed in set $\mathcal{A}$ (resp. $\mathcal{B}$). If two candidate programs P_i and P_j produce the same output for $\vec{x}$ (i.e. b_{ij} is *True*), then P_i and P_j must be placed in the same set. On the other hand, if two programs produce different outputs (i.e. b_{ij} is *False*), then only one of them can be placed in set $\mathcal{A}$.

Finally, each program must be placed in one and only one set. To enforce this, we add the constraints:

$$\bigwedge_{i,j\in\{1,\ldots,n\},i<j} b_{ij} \rightarrow \left((p_i^{\mathcal{A}} \wedge p_j^{\mathcal{A}}) \vee (p_i^{\mathcal{B}} \wedge p_j^{\mathcal{B}})\right), \tag{3.11}$$

$$\bigwedge_{i,j\in\{1,\ldots,n\},i<j} \neg b_{ij} \rightarrow (\neg p_i^{\mathcal{A}} \vee \neg p_j^{\mathcal{A}}), \text{ and} \tag{3.12}$$

$$\bigwedge_{i\in\{1,\ldots,n\}} p_i^{\mathcal{A}} \neq p_i^{\mathcal{B}}. \tag{3.13}$$

To ensure we have at least one program in set $\mathcal{A}$, we add the constraint:

$$\bigvee_{i\in\{1,\ldots,n\}} p_i^{\mathcal{A}}. \tag{3.14}$$

Ideally, we would like $\mathcal{A}$ and $\mathcal{B}$ to have the same number of elements, so that when the user classifies $\vec{x}$ as either correct or incorrect we can eliminate half the programs from the search space. To achieve this, we add the optimisation objective:

$$\text{min.} \left| \sum_{i=1}^{n} p_i^A - \sum_{i=1}^{n} p_i^B \right|. \tag{3.15}$$

As before, we use a MaxSMT solver to solve the formula. If the formula is unsatisfiable, then all programs are equivalent and there is no distinguishing input. In this case we can keep only one program and discard the rest. If the formula is satisfiable, then the model assignment for $\vec{x}$ is the distinguishing input. In this situation, the synthesizer shows $\vec{x}$ to the user, along with the output of any program in $\mathcal{A}$ run with input $\vec{x}$ (they all have the same output). Finally, we ask the user if this is the desired output for input $\vec{x}$. If the output is correct, we can eliminate from the search space all programs in $\mathcal{B}$. Otherwise, we remove all programs in $\mathcal{A}$.

3.5 Regex Synthesizers

In this section, we discuss prior work on the synthesis of regular expressions that is most closely related to our approach. Previous approaches that perform general string processing [22, 72] restrict the form of the regular expressions that can be synthesised. In contrast, we support a wide range of regular expressions operators, including the Kleene closure, positive closure, option, and range. More recent work that targets the synthesis of regexes is done by ALPHAREGEX [32] and REGEL [8].

ALPHAREGEX performs an enumerative search and uses under- and over-approximations of regexes to prune the search space. However, ALPHAREGEX is limited to the binary alphabet and does not support the kind of regexes that we need to synthesise for form validations. ALPHAREGEX is analysed in depth in Section 3.5.1.

REGEL is a state-of-the-art synthesizer of regular expressions based on a multi-modal approach that combines input-output examples with a natural language description of user intent. They use natural language to build sketches that capture the high-level structure of the regex to be synthesised and

subsequently use the input-output examples to fill those sketches. REGEL prunes the search space by using ALPHAREGEX-like under- and over-approximations and symbolic regexes combined with SMT-based reasoning.

There are other approaches that synthesise regexes solely from natural language [31, 34, 74]. We see these approaches as orthogonal to ours and expect that FOREST can be improved by hints provided by a natural language component such as was done in REGEL. To the best of our knowledge, no previous work focused on the synthesis of conditions over capturing groups.

3.5.1 ALPHAREGEX

In 2016, Lee et al. [32] presented ALPHAREGEX, a PBE synthesizer of regular expressions in the binary alphabet, $\Sigma = \{0, 1\}$. Like with FOREST, with ALPHAREGEX the user describes the desired accepted language by providing a set of positive and negative examples. The synthesised regular expression must define a language that includes all the positive examples and none of the negative ones. ALPHAREGEX uses an enumerative search technique, with a ranking method that prioritises simpler expressions.

The algorithm starts by examining the simplest regular expressions, `0` and `1`. If these are not consistent with the examples, it checks more complex productions such as `0|0`, `0|1`, `1|0`, `1|1` (i.e. expressions in the form of $\square|\square$), `00`, `01`, `10`, `11` (i.e. expressions in the form of $\square\square$), and `0`$*$, `1`$*$, (i.e. expressions in the form of $\square*$).

Here, we introduce the hole ($\square$), a placeholder for any regular expression. Regular expressions with or without holes are the states of the search. The algorithm generates regular expressions by iteratively replacing holes with other states and checking if the resulting regular expressions are correct.

To speed up the search, ALPHAREGEX makes use of three different kinds of search space pruning techniques: over-approximation, under-approximation and elimination of redundant states.

Over-approximation is achieved by replacing holes in the current state with $(0|1)*$. This regular expression describes the language that accepts all the strings that can be written using the binary alphabet. Therefore, over-approximation makes the state as general as it can be. If the over-approximation rejects at least one of the positive examples we conclude that this state can never be used to build a solution.

Example 3.5.1. Consider the state `1`$\square$, i.e., the concatenation of `1` with a hole, which can be filled with any regular expression. The over-approximation of this state, `1`$(0|1)*$, accepts all strings that a regular expression of the form `1`$\square$ could possibly accept. Thus, if `1`$(0|1)*$ does not accept all the positive examples, this state is not worth considering, and can be pruned.

Under-approximation consists in replacing holes in the current state with $\emptyset$, the regular expression that corresponds to the empty language. The concatenation of any regular expression r with $\emptyset$, $r\emptyset$, results in $\emptyset$. The union of any regular expression r with $\emptyset$, $r|\emptyset$, is the regular expression r itself. If the under-approximation does not reject any of the negative examples, we conclude that this state can never be used to build a correct solution.

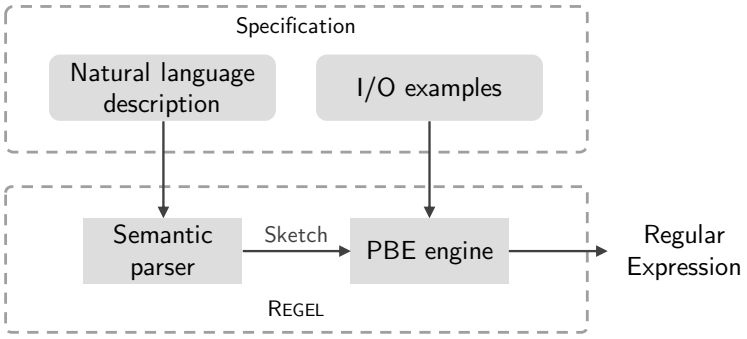


Figure 3.5: REGEL's synthesis pipeline

Example 3.5.2. Consider now the state $1|\square$, i.e., the union of 1 with a hole. The under-approximation of this state, $1|\emptyset = 1$, restricts $1|\square$ to the language that accepts as few strings as possible, in this case, the one containing only the string with a single '1'. If 1 does not reject any of the negative examples, then this state can be pruned.

Elimination of redundant states is done using the equivalences of regular expressions resulting from the algebraic rules of regular expressions. For example, the algorithm needs not evaluate the expression $s|r$ if it has already considered $r|s$, where r and s are regular expressions, as these expressions are equivalent.

The authors tested ALPHAREGEX in 25 benchmark problems collected from textbooks on automata theory which are provided along with the code for their tool. ALPHAREGEX proved capable of synthesising relatively complex expressions, with more than a dozen operations, in under one minute.

3.5.2 REGEL

In 2020, Chen et al. [8] proposed REGEL, a multi-modal sketch-based regex synthesizer. REGEL accepts two different kinds of description of desired behaviour: (i) a natural language description of the desired pattern (which is used to generate a sketch), and (ii) a set of positive and negative examples (which is used to complete the sketch). REGEL's synthesis pipeline is shown in Figure 3.5

REGEL's DSL is a lot more expressive than that of ALPHAREGEX. They synthesise common regex operators: concatenation, union, Kleene closure, positive closure, option and range. Besides the standard regex operators, they accept operations and (intersect) and not (complement), which are implemented at the automaton level.

Sketch generation. The first step of REGEL's synthesis is to produce a ranked list of sketches from the natural language description of the desired pattern using a semantic parser. Intuitively, REGEL's sketches represent a family of regexes that conform to a high-level structure. A sketch is then a program of an extended version of the DSL that includes a "constrained hole" construct, denoted $\square\{S_1, ..., S_n\}$, where $S_1, ..., S_n$ are sub-regexes. Specifically, regex r belongs to the space of regexes defined by $\square\{S_1, ..., S_n\}$ if one of the leaf nodes of r is one of the S_i sub-regexes. In addition to constrained holes, REGEL's

sketches can contain operators in their regex DSL. For example, a sketch can be of the form `f`$(S_1, ..., S_n)$ where `f` denotes a DSL operator.

Example 3.5.3. Consider the following REGEL sketch: S_1 = □{`[0-9]{7}`, `-`}. The regular expression r_1=`(([A-Za-z]|-){2}|[0-9]{7}){1,4}` can result from S_1 because one of its leaf nodes, `-`, is in the constrained hole. Consider now S_2 = □{`[0-9]{3}`,`[0-9]{2}`,`-`,`[0-9]{4}`} and r_2 = `([0-9]|-){11}`. Similarly, r_2 can result from S_2.

Sketch completion. Then, given a sketch S, REGEL's PBE engine tries to find a concrete regex that is both a valid completion of S and consistent with the input-output examples using sketch-guided enumerative search. Their sketch completion procedure leverages two main ideas.

First, like ALPHAREGEX, they use lightweight deductive reasoning to prune infeasible partial regexes by constructing over- and under- approximations. However, unlike ALPHAREGEX, they are able to build these approximations using the hints obtained from the natural language and therefore perform more precise reasoning.

Second, they use symbolic regexes to prune large parts of the search space. When synthesising constructs that take integer constants as arguments (like the range operator), they use a symbolic integer κ that represents any possible integer rather than explicitly enumerating possible integer values. Then, they generate a SMT formula that over-approximates the concrete regex. In the end, the concrete values of κ are extracted from the resulting SMT model.

Chen et al. evaluate their approach on 322 regexes and showed that their multi-modal approach can successfully synthesise the intended regex in 80% of the cases. In comparison, using only natural language can solve 43% of these benchmarks and an example-only baseline can solve only 26%. They show also that their PBE engine is an order of magnitude faster than ALPHAREGEX.

Chapter 4

Regular Expression Synthesis

In this chapter, we describe the first stage of FOREST's synthesis procedure, which produces the first component of the regex validation: a regular expression that matches all valid examples and none of the invalid examples. This regular expression serves as basis for the synthesis of the second and third components of the regex validation.

Before the synthesis procedure starts, we define the set of operators that can be used to build the desired regex, as well as the values each operator can take as argument, i.e., FOREST's DSL. FOREST builds its DSL based on the user-provided examples. Each DSL is dynamically constructed to fit the problem at hand: it is as restricted as possible, without losing any expressiveness necessary to ensure it includes the correct regex. The DSL construction procedure is detailed in Section 4.1.

Following the example of several state-of-the-art synthesizers [15, 17, 32, 45, 47, 58], our synthesis algorithm employs enumerative search, as introduced in Section 2.3.3. The enumeration cycle finds regexes that are consistent with the user-provided examples. In order to choose an expression among those generated by the enumeration, FOREST interacts with the user. The result of each interaction is a new input-output example that strengthens the original specification. Both the enumeration and the interaction cycles are depicted in Figure 4.1.

As mentioned before, input-output examples are an ambiguous specification. Thus, FOREST does not return the first regex consistent with the examples that it finds, as it may not correspond to the user's intended behaviour. Instead, FOREST provides an interaction model. The interactive model is further described in Section 4.3.

4.1 Domain Specific Language

Before the synthesis procedure starts, the DSL of the enumerated regexes must be defined. This includes the definition of the operators that can be used to build the desired regex, as well as the values each operator can take as argument.

FOREST aims at producing a regular expression that fully matches the strings provided as valid examples. We consider a successful match only when the regular expression matches the entire string.

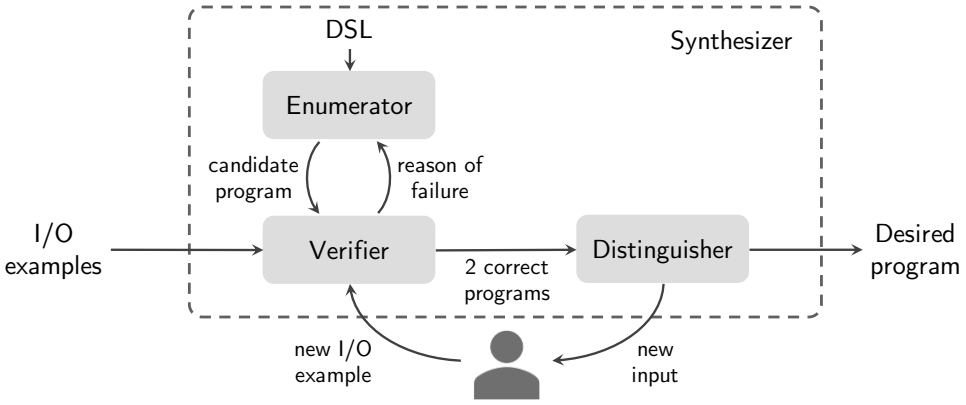


Figure 4.1: Interactive enumeration-based synthesis

To construct the regex, the DSL always includes the union and concatenation operators. As introduced in Section 2.1.2, the union of two arbitrary regexes r and s creates a new regex, denoted $r|s$. A string matches $r|s$ if it matches r or s. Similarly, the concatenation of any two regexes r and s, creates a new regex, rs. If a string p is matched by r and a string q is matched by s, then pq is matched by rs.

In addition, we consider several regular expression quantifiers. A quantifier can be applied to any regex r and is equivalent to concatenating r with itself a certain number of times. The DSL includes the following quantifiers:

- *Kleene* closure (*), which matches 0 or more times,
- positive closure (+), which matches 1 or more times,
- option (?), which matches 1 or 0 times,
- range ($\{m\}$), which matches exactly m times, and
- range ($\{m, n\}$), which matches at least m times and at most n times.

The possible values for the range operators are limited depending on the valid examples provided by the user. For the single-valued range operator, $\{m\}$, we consider only the integer values such that $2 \leq m \leq l$, where l is the length of the longest valid example string. Note that $m = 1$ is not considered since it is the identity function ($r\{1\} = r$). In the two-valued range operator, $\{m, n\}$, the values of m and n are limited to integers such that $0 \leq m < n \leq l$, where l is again the length of the longest valid example string. The tuple (0,1) is not considered, since its semantics are equivalent to that of the option quantifier: $r\{0, 1\} = r?$.

All operators can be applied to regex literals or composed with each other to form more complex expressions. The regex literals considered in the synthesis procedure also depend on the valid input examples. These include the individual letters, digits or symbols present in the examples and all character classes that include them. The character class `[`r_1`-`r_n`]` is shorthand for the regular expression $r_1|r_2|\ldots|r_n$, when $r_1r_2...r_n$ form a logical sequence. The character classes contemplated in the DSL are `[0-9]`, `[A-Z]`, `[a-z]` and all combinations of those, such as `[A-Za-z]` or `[0-9A-Za-z]`. Additionally, `[0-9A-F]` and `[0-9a-f]` are used to represent hexadecimal numbers.

$$
\begin{aligned}
Re &\rightarrow \texttt{concat}(Re, Re) \mid \texttt{union}(Re, Re) \mid \texttt{kleene}(Re) \mid \texttt{posit}(Re) \\
&\mid \texttt{option}(Re) \mid \texttt{range}(Re, RangeLit) \mid \texttt{[0-9]} \mid \texttt{/} \\
RangeLit &\rightarrow 2 \mid 3 \mid ... \mid 10 \mid (0,2) \mid (1,2) \mid (0,3) \mid ... \mid (8,10) \mid (9,10)
\end{aligned}
$$

Figure 4.2: CFG that represents the DSL of regular expressions for the motivating example in Section 1.1. Re is the start symbol and the representation of the regex type, $RangeLit$ represents the possible values for the argument of the range operator.

As mentioned in Section 2.3.2, the DSL can be represented as a CFG. The CFG's terminal symbols Σ include the names of all regex operators, as well as the regex literals and the range operator values. N, the non-terminal symbols, are the data types in the DSL.

Example 4.1.1. The literal values on the DSL depend on the input examples. Recall the motivating example in Section 1.1, where the valid input values were:

19/08/1996	22/09/2000	29/09/2003
26/10/1998	01/12/2001	31/08/2015

In this scenario, the length of the longest valid example is $l = 10$, so the range value literals are $m \in \{2, ..., 10\}$, for the single valued range, and $(n, m) : 0 \leq n < m \leq 10$ for the two-valued range. The characters in the examples are '/', which becomes itself a regex literal, and digits, which introduce the character class `[0-9]`. The CFG is then defined as shown in Figure 4.2.

4.2 Enumeration

To enumerate regexes, FOREST requires a structure capable of representing every feasible expression. Orvalho et. al. [45] describe two different representations: tree-based and line-based. The tree-based representation follows a typical hierarchical composition of operations, as usually seen in functional languages, while the line-based representation emulates the structure of a program written in an imperative language, where each operation is seen as a line of code.

We opt to use a tree-based representation of the search space, using k-trees. A k-tree of depth d is a tree in which every internal node has exactly k children and every leaf node is at depth d. If k is the greatest arity among all DSL constructs, then a k-tree of depth d can represent all programs of depth up to d in that DSL. For that, a symbol of the DSL is assigned to each node and the tree can be interpreted as a program. In any regex DSL built by FOREST, the maximum arity is always 2, so all regexes in the search space can be represented using 2-trees. In Figure 4.3, the regex from the motivating example in Section 1.1, `[0-9]{2}/[0-9]{2}/[0-9]{4}`, is represented as a 2-tree of depth 5.

To explore the search space in order of increasing complexity, the k-tree enumerator starts by analysing trees of lower depths and progressively increases the depth of the tree as trees of previous depths are exhausted. This way we ensure the first regex found is of the smallest depth possible.

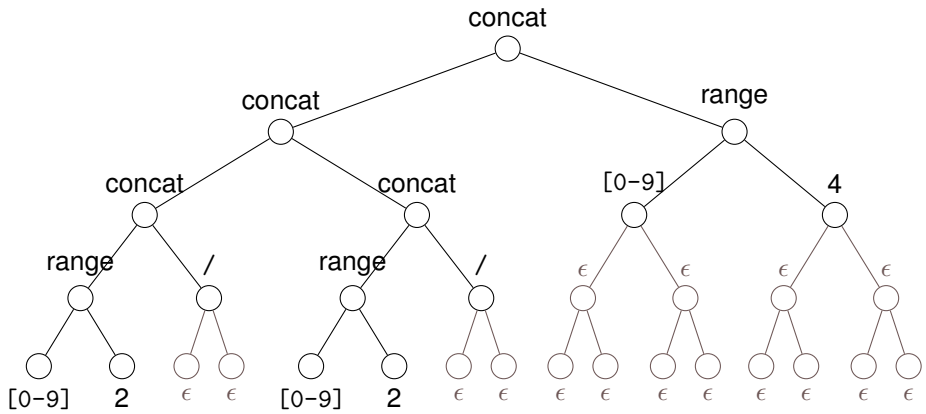


Figure 4.3: `[0-9]{2}/[0-9]{2}/[0-9]{4}` represented as a k-tree.

To explore the space of trees, the enumerator encodes the k-tree as an SMT formula. The constraints in the SMT formula ensure that the program is well-typed. A model that satisfies the formula represents an assignment of a DSL symbol to each tree node. This assignment is then used to build a valid regex.

4.2.1 *K*-tree Encoding

In this section we offer a formal description of the k-tree encoding, as proposed by Chen et. al. [9, 45]. A k-tree is a tree where all leaves are at the same depth and with constant branching factor k. Consider a k-tree of depth d. Assume each node in the k-tree is assigned a unique positive integer index consistent with level-order traversal. Let N be the set all nodes' indices, $N = \{1, 2, ..., k^d - 1\}$. Let L be the set of indices that correspond to leaf nodes and I the set of indices of internal nodes. Thus $N = I \cup L$. Furthermore, let $C(i)$ represent the indices of all children of node i.

Let $id : \Sigma \to \mathbb{N}$ be a one-to-one mapping of each terminal symbol (i.e., operators and literals) in the DSL to a positive integer. Because the arity of some DSL operations is smaller than k, there are some children nodes that cannot be assigned any symbol. We introduce the empty symbol, ϵ, with $id(\epsilon) = 0$, which is assigned to nodes without symbols.

Encoding variables. We define an integer variable n_i for all $i \in N$. Assigning $n_i = id(k)$ means that we assign the symbol k to node i.

Next, we add constraints over these variables that ensure the generated regular expression is well-typed. All examples in the remainder of this section refer to the DSL presented in Example 4.1.1 built for the the motivating example described in Section 1.1.

Leaf constraints. Because they have no children, the leaf nodes must be empty or assigned to a literal of the DSL. Let T be the union of ϵ with the set of terminal symbols in the DSL that correspond to literals.

$$\bigwedge_{i \in L} \bigvee_{p \in T} n_i = id(p) \tag{4.1}$$

Example 4.2.1. For each leaf node i we add the constraint:

$$n_i = id(\texttt{[0-9]}) \vee n_i = id(\texttt{/}) \vee n_i = id(2) \vee ... \vee n_i = id(10) \vee n_i = id((0,2)) \vee$$
$$\vee n_i = id((1,2)) \vee n_i = id((0,3)) \vee ... \vee n_i = id((8,10)) \vee n_i = id((9,10))$$

Children constraints. If a DSL symbol p is assigned to a node i, the children of i, $C(i)$, must have types consistent with the parameters of p. For each $p \in \Sigma$ and $j \in \{1, ..., k\}$, where k is the largest arity among all DSL constructs, we define $T(p, j)$. If p corresponds to an operator, $T(p, j)$ denotes the type of parameter j of p. If $j > arity(p)$, then $T(p, j) = \epsilon$. If p represents a literal, then $T(p, j) = \epsilon$ for every j. Finally, let $\Sigma(T(p, j)) \subset \Sigma$ be the set of terminal symbols in the DSL that have type $T(p, j)$.

$$\bigwedge_{p \in \Sigma, i \in I} \left(n_i = id(p) \Rightarrow \bigwedge_{j \in C(i)} \bigvee_{t \in \Sigma(T(p,j))} n_j = id(t) \right) \tag{4.2}$$

Example 4.2.2. If node 1 is assigned `range`, then its children, nodes 2 and 3, must be assigned symbols of types consistent with its parameters. $T(\texttt{range}, 1) = Re$, so n_2 must be assigned one of the symbols in $\Sigma(Re) = \{\texttt{union}, \texttt{concat}, \texttt{kleene}, \texttt{posit}, \texttt{option}, \texttt{range}, \texttt{[0-9]}, \texttt{/}\}$. $T(\texttt{range}, 2) = RangeLit$, so n_3 must be assigned one of the symbols in $\Sigma(RangeLit) = \{2, ..., 10, (0,2), (1,2), ..., (9,10)\}$. To enforce this, we add the following constraint:

$$\begin{aligned} n_1 = id(\texttt{range}) \Rightarrow (n_2 = id(\texttt{union}) \vee n_2 = id(\texttt{concat}) \vee n_2 = id(\texttt{kleene}) \vee \\ \vee n_2 = id(\texttt{posit}) \vee n_2 = id(\texttt{option}) \vee n_2 = id(\texttt{range}) \vee \\ \vee n_2 = id(\texttt{[0-9]}) \vee n_2 = id(\texttt{/})\,) \\ \wedge (n_3 = id(2) \vee ... \vee n_3 = id(10) \vee n_3 = id((0,2)) \vee \\ \vee\ n_3 = id((1,2)) \vee ... \vee n_3 = id((9,10))\,) \end{aligned}$$

Output constraints. The root node must be assigned a DSL symbol consistent with the output type. Re is the desired output type in our DSL. Then $\Sigma(Re)$ is the set of symbols in the DSL that have type Re.

$$\bigvee_{p \in \Sigma(Re)} n_1 = id(p) \tag{4.3}$$

Example 4.2.3. We want or regular expression to have the DSL type Re, so the output constraint for our domain is

$$n_1 = id(\texttt{union}) \vee n_1 = id(\texttt{concat}) \vee n_1 = id(\texttt{kleene}) \vee n_1 = id(\texttt{posit}) \vee$$
$$\vee n_1 = id(\texttt{option}) \vee n_1 = id(\texttt{range}) \vee n_1 = id(\texttt{[0-9]}) \vee n_1 = id(\texttt{/})$$

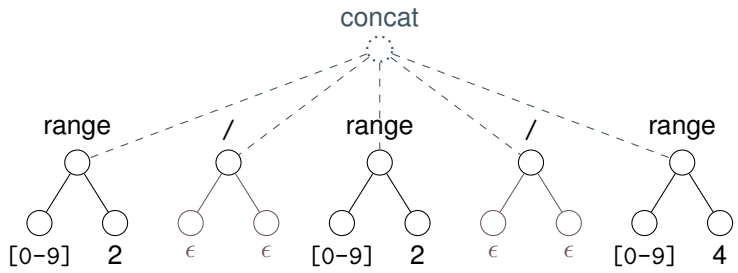


Figure 4.4: `[0-9]{2}/[0-9]{2}/[0-9]{4}` represented as a multi-tree, resulting from the concatenation of 5 simpler regexes.

4.2.2 Multi-tree Representation

We considered several validators for digital forms and observed that many regexes in this domain are the concatenation of relatively simple regexes. However, the successive concatenation of simple regexes quickly becomes complex in its k-tree representation. Recall the regex for date validation presented in the motivating example in Section 1.1: `[0-9]{2}/[0-9]{2}/[0-9]{4}`. Even though this regular expression is the concatenation of 5 simple sub-expressions, each of which can be represented using a tree of depth at most 2, its representation as a k-tree requires a tree of depth 5, as shown in Figure 4.3.

The idea behind the multi-tree constructs is to allow the number of concatenated sub-expressions to grow without it reflecting exponentially on the encoding. The multi-tree structure is constituted by n k-trees, whose roots are connected by an artificial top-level root node. The root-node is "assigned" an n-ary concatenation operator, whose semantics is analogous to that of the binary concatenation of regexes described in Section 4.1. Since the return type of `concat` is the same as that of its parameters, Re, all the root nodes have the same type as they did in the original k-tree encoding. Because the root-node has a fixed assignment, it is not included in the SMT encoding, and it is artificially added to the model that satisfies it.

Having a top-level concatenation node that connects n k-trees of depth d, we are able to represent regexes using fewer nodes. Figure 4.4 shows the multi-tree representation of the same regex as Figure 4.3, and shows that the multi-tree construct can represent this expression using half the nodes.

In the k-tree encoding, an increase in the depth d of the tree corresponds to an increase in the expression's complexity. Thus, the enumerator successively explores k-trees of increasing depth. However, multi-tree has 2 measures of complexity: the depth of the trees, d, and the number of trees, n. Note that, since it is not encoded into SMT, the root-node level is not taken into account for the depth of the multi-tree. FOREST employs two different methods for increasing these values: static multi-tree and dynamic multi-tree.

Static multi-tree

In the static multi-tree method, FOREST fixes n and progressively increases d. To find the value of n, there is a preprocessing step, in which FOREST identifies patterns in the valid examples. This is done by first identifying substrings common to all examples. A substring is considered a dividing substring if

it occurs exactly the same number of times in all examples. Dividing substrings must also occur in the same order in all examples. Then, we split each example before and after the dividing substrings. Each example becomes an array of n strings.

Example 4.2.4. Consider the valid examples from the motivating example in Section 1.1. In these examples, '/' is a dividing substring because it occurs in every example, and exactly twice in each one. '0' is a common substring but not a dividing substring because it does not occur the same number or times in all examples. After splitting on '/', each example becomes a tuple of 5 strings:

('19', '/', '08', '/', '1996')	('22', '/', '09', '/', '2000')	('29', '/', '09', '/', '2003')
('26', '/', '10', '/', '1998')	('01', '/', '12', '/', '2001')	('31', '/', '08', '/', '2015')

Then, we apply the multi-tree method with n trees. If, for every $i \in \{1, ..., n\}$, the i^{th} sub-tree represents a regex that matches all strings in the i^{th} position of the example arrays, then the concatenation of the n regexes matches the original example strings. Since each tree is only synthesizing for a part of the original input strings, a reduced DSL can be recomputed for each tree.

Example 4.2.5. Recall the split examples from Example 4.2.4, and the DSL built before the split shown in Figure 4.2. This DSL has two regex literals, `/` and `[0-9]`. However, we can see that only `[0-9]` is needed in the 1st, 3rd and 5th splits. Moreover, since the length of the splits is smaller than 10, fewer values for $RangeLit$ need to be considered. We can build 5 reduced DSLs, $\{D_1, D_2, D_3, D_4, D_5\}$, where each D_i includes only the required symbols to build the regex that matches the i^{th} split. $D_1 = D_3$ are represented as:

$$\begin{aligned} Re &\rightarrow \texttt{concat}(Re, Re) \mid \texttt{union}(Re, Re) \mid \texttt{kleene}(Re) \mid \texttt{posit}(Re) \\ &\mid \texttt{option}(Re) \mid \texttt{range}(Re, RangeLit) \mid \texttt{[0-9]} \\ RangeLit &\rightarrow 2 \mid (0, 2) \mid (1, 2) \end{aligned}$$

Analogously, `[0-9]` is never needed in the 2nd and 4th splits. $D_2 = D_4$ are represented as:

$$\begin{aligned} Re &\rightarrow \texttt{concat}(Re, Re) \mid \texttt{union}(Re, Re) \mid \texttt{kleene}(Re) \\ &\mid \texttt{posit}(Re) \mid \texttt{option}(Re) \mid \texttt{/} \end{aligned}$$

In the 2nd and 4th splits, the maximum length is 1, which means there are no possible values for the range operator. Thus, `range` is not part of D_2 and D_4. D_5 is similar to D_1 and D_3 but it has more possible values for `range`, since the maximum length is 4. Now, each of the 5 k-trees can use the corresponding D_i instead of the whole original DSL.

Dynamic multi-tree

FOREST employs the dynamic multi-tree method when the examples cannot be split because there are no dividing substrings. In this scenario, the enumerator still uses a multi-tree construct to represent the

regex. However, the number of trees is not fixed and all trees use the original, complete DSL. FOREST varies the complexity of the enumerated expressions by increasing either the number of trees n or their depth d. A multi-tree structure with n k-trees of depth d has $n \times (k^d - 1)$ nodes. Different combinations of (n, d) are tested in increasing order of number of nodes, starting with $n = 1$ and $d = 2$, which is equivalent to a simple k-tree of depth 2.

4.2.3 Pruning

The tree-based representation is especially advantageous when pruning the search space, as it facilitates the propagation of constraints from any node in the tree to the leaves. FOREST applies several pruning techniques that leverage properties of regular expressions. FOREST's pruning can be divided into two kinds. The first kind refer to properties of the DSL that remain unchanged during the synthesis procedure and are added at the beginning, along with the encoding constraints.

Like in most other languages, there can be multiple regexes that, although different, have the same semantics. Here are some examples of such equivalences:

$$(r*)* \equiv r* \qquad (r?)? \equiv r? \qquad (r+)+ \equiv r+$$
$$(r+)* \equiv (r*)+ \equiv r* \qquad (r?)* \equiv (r*)? \equiv r* \qquad (r?)+ \equiv (r+)? \equiv r*$$
$$(r*)\{m\} \equiv (r\{m\})* \qquad (r+)\{m\} \equiv (r\{m\})+ \qquad (r?)\{m\} \equiv (r\{m\})?$$
$$r\{n\}\{m\} \equiv r\{m\}\{n\} \equiv r\{m \times n\}$$

During the synthesis procedure, we want to avoid enumerating several equivalent expressions. To achieve this, we add SMT constraints that block all but one possible representation of each regex. Take, for example, $(r?)+ \equiv r*$. We only want one way to represent this regex, so we block the construction $(r?)+$ for any regex r. Let I be the set of indices of all internal nodes of the trees and $Ch_L(i)$ denote the index of the left child of node i. We add the constraint:

$$\bigwedge_{i \in I} n_{Ch_L(i)} = id(\texttt{option}) \Rightarrow n_i \neq id(\texttt{posit}) \tag{4.4}$$

The remaining equivalences shown above are implemented analogously.

Another relevant rule in the DSL of regular expressions is the idempotence of union: $r|r = r$. To prevent the enumeration of expressions of the type $r|r$, every time the `union` operator is assigned to a node i, we force the sub-tree underneath i's left child to be different from the sub-tree underneath i's right child by at least one node. For each internal node $i \in I$, let L_i (resp. R_i) be an ordered list of the indices of all nodes in i's left (resp. right) child sub-tree, i.e., the sub-tree whose root is i's left (resp. right) child. We add the following constraint to ensure the parameters of `union` are different:

$$\bigwedge_{i \in I} \left(n_i = id(\texttt{union}) \Rightarrow \bigvee_{1 \leq j \leq |L_i|} n_{L_i[j]} \neq n_{R_i[j]} \right) \tag{4.5}$$

The second kind of pruning constraints are added during enumeration. These do not refer directly to properties of regexes, but rather to characteristics of the current instance. When a regex that is not consistent with the examples is enumerated, it is eliminated from the search space. Let $m : \mathcal{N} \to \mathbb{N}$ be a model to the formula resulting from the encoding of the trees, i.e., an assignment of each SMT variable, n_i, to a DSL symbol's identifier. For each encoding variable n_i, $m(n_i) = id(p)$ means that p was the DSL symbol assigned to node i in the previous solver call. Thus, we block the current regex by adding the constraint:

$$\bigvee_{i \in N} n_i \neq m(n_i). \tag{4.6}$$

Along with the incorrect regex, we want to eliminate regexes that are equivalent to it. The union operator in the regular expressions DSL is commutative: $r|s = s|r$, for any regexes r and s. Thus, whenever an expression containing $r|s$ is discarded, we eliminate the expression that contains $s|r$ in its place as well. Let U contain all the indices of the nodes that were assigned the `union` operator in the previous solver call: $U = \{i \in N : m(n_i) = id(\texttt{union})\}$. First, we block the assignment of all the nodes in the tree that are not descendant from the `union` node. Then, we prevent the variables on i's left child sub-tree from being assigned the equivalent model value on the right sub-tree, and vice-versa:

$$\bigwedge_{i \in U} \left(\bigvee_{i \in N \setminus (L_i \cup R_i)} n_i \neq m(n_i) \ \vee \bigvee_{1 \leq j \leq |L_i|} n_{L_i[j]} \neq m(n_{R_i[j]}) \ \vee \bigvee_{1 \leq k \leq |R_i|} n_{R_i[k]} \neq m(n_{L_i[k]}) \right) \tag{4.7}$$

Besides expressions that are equivalent to the incorrect enumerated expression, we also remove from the search space other expressions that fail in the same way. The goal of quantifiers `kleene`, `posit` and `range` is to allow a regex to be matched more than once. If `kleene` is assigned to the root of one of the k-trees, this tree represents $r*$, where r is an arbitrary regex. We test all valid examples for the presence of $r\{2\}$. If $r\{2\}$ is not present in any of the examples, we block the expression $r*$. The same procedure is followed for `posit` and `range`.

4.2.4 Sketch-based

As introduced in Section 3.1, some state-of-the are synthesizers use sketches to speed up their enumerative search. Different methods are used to generate and complete sketches. In this section we describe a few of them, which we implemented in FOREST, though they are disabled by default.

Sketch generation

When enumerating sketches, the procedure is very similar to the one used to enumerate programs: we have a multi-tree construct and successively enumerate sketches using the k-tree encoding for each sub-tree. The main difference between sketch enumeration and the usual regex enumeration procedure lies in the DSL: we adapt the DSL to produce incomplete programs (with holes) instead of complete regexes. We remove literal values and constants from the DSL. We introduce two types of "hole" to replace them: the regex-hole $\square_{re}$ and the range-hole $\square_{range}$. Then, each sketch can be completed with

$$Re \rightarrow \texttt{concat}(Re, Re) \mid \texttt{union}(Re, Re) \mid \texttt{kleene}(Re) \mid \texttt{posit}(Re)$$
$$\mid \texttt{option}(Re) \mid \texttt{range}(Re, \square_{range}) \mid \square_{re}$$

Figure 4.5: CFG that represents the sketch DSL of regular expressions for the motivating example in Section 1.1.

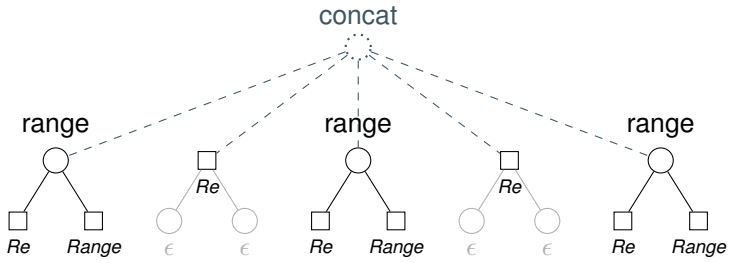


Figure 4.6: Sketch multi-tree whose completion results in `[0-9]{2}/[0-9]{2}/[0-9]{4}`.

concrete regex and range literals to get a complete regular expression.

Example 4.2.6. Recall the DSL built for the motivating example in Section 1.1 shown in Figure 4.2. If we were synthesising this instance with sketch-based enumeration, we would have the DSL shown in Figure 4.5, where the regex literals have been replaced with regex-holes, $\square_{re}$, and the range values with range-holes, $\square_{range}$.

Note that we enumerate fewer sketches than we enumerate programs during the standard enumeration procedure, because each sketch corresponds to many concrete programs. In Figure 4.6, we can see the sketch that has to be enumerated during sketch-enumeration in order to synthesise `[0-9]{2}/[0-9]{2}/[0-9]{4}` during sketch-completion.

Sketch completion

To perform sketch-completion, we need to bring back into consideration the information we removed from the DSL during sketch-enumeration, i.e., the possible literal values that can be used to fill each hole in our sketch.

Example 4.2.7. Consider, once again, the motivating example described in Section 1.1. If we are synthesising this instance using sketch-based enumeration, the following domains are considered for each hole in the sketches:

- $\mathcal{D}(\square_{re})$ = {`[0-9]`, `/`}
- $\mathcal{D}(\square_{range})$ = {2, 3, ..., 10, (0,2), (1,2), (0,3) , ..., (8,10), (9,10)}

Several methods can be used to find the correct values to fill the holes in a sketch within the respective domain. In FOREST, we tried two different approaches: graph-based and SMT-based sketch completion.

Graph-based completion is implemented as described by Ramos in his MSc. Thesis [53]. We start by identifying all possible complete programs resulting from a sketch. To do this, we first consider the root node and recursively visit all its descendants, traversing the sketch tree in Depth-First Search (DFS) order. For each tree node we visit, if it is an internal node, we visit its descendants. If it is a hole, we return a list of all possible completions for it, i.e., the values in its corresponding domain. These values are stored in the parent node. In the end of the traversal, each internal node has a list of all possible completions for all holes in its descendant sub-tree. The root node contains information about all completion values for all holes in the tree. Then, the complete programs resulting from the sketch are simply the Cartesian product of all domains stored in the root node. Each concrete program is built and tested against the specification by the *verifier*, and the synthesizer stores those that satisfy the specification.

SMT-based completion tries to use logic-based reasoning during the completion. We do not seek *all* possible completions for a sketch, but rather a set of completions that satisfy our specification. In a way, we are pulling the *verifier*'s role into the sketch completion procedure. The implementation of this method requires an SMT solver with a regex theory, such as Z3 [6, 12]. The first step is the same as Graph-based completion: traverse the sketch tree and get the possible domains for each hole.

Ideally, we want to assign to each hole in the sketch an SMT variable of type regex, whose domain is limited according to that hole's domain. Then, we would write the sketch regex in SMT using these variables. Finally, we add constraints that state the desired regex must match all valid examples and none of the invalid. If the formula is unsatisfiable, then the sketch cannot be completed in a way that satisfies the specification, and can be discarded. If the formula is satisfiable, the resulting model results in a completion of the sketch that is consistent with the specification. However, Z3's regex theory is not sufficiently expressive to allow this method's implementation: there cannot be variables inside an SMT regular expression. Therefore, our SMT-based completion is done as follows:

For each sketch S, let $\mathcal{S}^*$ be the set of all regexes that result from the completion of S. This set is obtained from the Cartesian product of all holes' domains, as before. For each regex $r_i \in \mathcal{S}^*$ we define an SMT predicate, ρ_i, such that, for any input string s, $\rho_i(s)$ is *True* if and only if regex r_i matches s. Let $\mathcal{V}$ be the set of all valid examples and $\mathcal{I}$ the set of all invalid examples. For each $i \in \{1, ..., |\mathcal{S}^*|\}$, we define a Boolean variable m_i that is assigned *True* if and only if the respective r_i matches all valid examples, $\mathcal{V}$, and none of the invalid examples, $\mathcal{I}$:

$$\bigwedge_{i \in \{1,\ldots,|\mathcal{S}^*|\}} m_i \leftrightarrow \left(\bigwedge_{x \in \mathcal{V}} \rho_i(x) \wedge \bigwedge_{x \in \mathcal{I}} \neg \rho_i(x) \right). \tag{4.8}$$

We want to find a regex r_i whose m_i is *True*. Thus, we add the constraint:

$$\bigvee_{i \in \{1,\ldots,|\mathcal{S}^*|\}} m_i \tag{4.9}$$

If this formula is unsatisfiable, there is no possible completion of sketch S that matches the specific-

ation. If it is satisfiable, then we store the regexes r_i whose m_i are assigned *True* in the resulting model. All these are correct regexes.

4.3 User Interaction

To solve possible ambiguities in the user-provided input-output examples, we implemented in FOREST two different interaction models: conversational clarification, as described in Section 3.4.1 and a variation of Ramos's interaction models, as described in Sections 3.4.2 and 3.4.3. By default, FOREST disambiguates the specification using conversational clarification.

4.3.1 Conversational Clarification

To increase confidence in FOREST's solution, FOREST disambiguates the specification by interacting with the user. We employ the interaction method described by Mayer et. al. [40], which has been successfully used in several synthesizers [33, 70, 71]. Upon finding two regexes that satisfy the user-provided examples, r_1 and r_2, FOREST computes a new input that is matched by one of the regexes but not by the other: a *distinguishing input*. To produce a distinguishing input, we use an SMT solver with a regex theory, such as Z3 [6, 12], to create two SMT predicates ρ_1 and ρ_2 such that, for any input string s, $\rho_1(s)$ (resp. $\rho_2(s)$) evaluates to *True* if and only if r_1 (resp r_2) matches s. Then, we use the SMT solver to solve the formula:

$$\exists s : \rho_1(s) \neq \rho_2(s). \tag{4.10}$$

If (4.10) is satisfiable, the string s that satisfies it is a distinguishing input. Once found, FOREST asks the user to classify this input as valid or invalid. The distinguishing input along with its correct output form a new input-output pair which is added to the specification, effectively eliminating either r_1 or r_2 from the search space. If, on the other hand, (4.10) is unsatisfiable then r_1 and r_2 are equivalent; FOREST keeps the shortest regex and discards the other.

After the first interaction with the user, the synthesis procedure continues only until the end of the current depth and number of trees. Therefore, if the desired expression requires a larger depth than that of the first enumerated expression that satisfied the examples, then FOREST never enumerates the desired expression and the returned expression will not satisfy the user's needs in corner cases not covered by the initial examples.

Example 4.3.1. Consider, for example, that FOREST enumerated two regexes consistent with the examples in the motivating example in Section 1.1: r_1 = `[0-9]{2}/[0-9]{2}/[0-9]{3,4}` and r_2 = `[0-9]{2}/[0-9]{2}/[0-9]{4}`. A possible distinguishing input between r_1 and r_2 returned by FOREST is s = "00/24/404". This input would be shown to the user. Since it does not represent a valid date, the user would classify s as invalid. Thus, "00/24/404" would be added to the set of invalid examples, and r_1, which matches s, would be eliminated from the search space.

4.3.2 Multi-distinguish

Besides the basic conversational clarification model, we extended FOREST with a new interaction model, based on the interaction models first introduced in 2020 by Ramos et al. [54]. As mentioned in Section 4.3, Ramos et al. present two interaction models: Y/N and OPTIONS. In the domain of regular expressions, the output is a Boolean value: either the string is matched by the regular expression or it is not. In practice, this entails a simplification of the Y/N and OPTIONS models, and they converge into a single model, which we call multi-distinguish interaction model.

The goal is to distinguish more than two regular expressions at once. Instead of triggering an interaction cycle right after finding two correct regular expressions, we continue the enumeration process past the second expression, until we have n correct expressions. Then, we disambiguate those n expressions at once, in order to generate fewer, more comprehensive distinguishing inputs, thus reducing the number of necessary queries to the user.

To disambiguate n regular expressions, $r_1, ..., r_n$, we start, as before, by defining n SMT predicates, $\rho_1, ..., \rho_n$ such that, for any input string s, $\rho_i(s)$ evaluates to *True* if and only if r_i matches s. We want to find a distinguishing string s that separates the regexes in two disjoint sets, $\mathcal{A}$ and $\mathcal{B}$, such that all regexes in $\mathcal{A}$ match s and none of the regexes in $\mathcal{B}$ matches s. Ideally, we want $\mathcal{A}$ and $\mathcal{B}$ to have the same number of regexes, so we can eliminate at least half the candidate expressions with each interaction.

To ensure at least two regexes in $r_1, ..., r_n$ produce a different result when matched against s, we add the hard constraint:

$$\bigvee_{i,j \in \{1,...,n\}, i<j} \rho_i(s) \neq \rho_j(s). \tag{4.11}$$

Then, we want to split the regexes into two sets $\mathcal{A}$ and $\mathcal{B}$ of approximately the same size. This problem can be modelled as a bipartite graph. Let the regexes be the vertices in the graph, and let there be an edge between r_i and r_j if and only if r_i and r_j have a different matching result to string s. Then the graph's sets correspond to sets $\mathcal{A}$ and $\mathcal{B}$. The resulting structure is a complete bipartite graph, a bipartite graph where each vertex of the first set is connected to every vertex of the second set.

Now, we want to build a bipartite graph such that both sets of vertices have the same size. Since the number of edges in the bipartite graph is $|\mathcal{A}| \cdot |\mathcal{B}|$, this is equivalent to maximising the number of edges in the graph, i.e., the number of pairs (r_i, r_j) which have different matching results to s.

Thus, to encode the goal of $\mathcal{A}$ and $\mathcal{B}$ having the same size, we add the soft constraints:

$$\bigwedge_{i,j \in \{1,...,n\}, i<j} \rho_i(s) \neq \rho_j(s). \tag{4.12}$$

We use a MaxSMT solver to find a model to this formula. If (4.11) is unsatisfiable, then all regexes in $r_1, ..., r_n$ are equivalent and we can keep only one of them. Otherwise, if the formula is satisfiable, s is a distinguishing string. Once found, FOREST follows a similar procedure to that described in Section 4.3.1: we ask the user to classify s as valid or invalid. Then, FOREST adds the example to the respective set, eliminating from the search space all regexes which are not in accordance to the user's answer.

Example 4.3.2. Suppose FOREST enumerates the following four regular expressions:

r_1 = `[0-9]{2}/[0-9]{0,2}/[0-9]{3,4}`,

r_2 = `[0-9]{2}/[0-9]{0,2}/[0-9]{4}`,

r_3 = `[0-9]{2}/[0-9]{2}/[0-9]{4}`, and

r_4 = `[0-9]{2}/[0-9]{2}/[0-9]{3,4}`.

A possible distinguishing input obtained using multi-distinguish is s = '01//2000'. s is accepted by r_1 and r_2 ($\mathcal{A} = \{r_1, r_2\}$) but not by r_3 or r_4 ($\mathcal{B} = \{r_3, r_4\}$). If we show s to the user, the user classifies it as invalid because it does not correspond to a valid date. We add this example to the set of invalid examples, thus eliminating r_1 and r_2 from the search space.

Chapter 5

Capturing Groups Synthesis

Aside from being a very expressive way to describe patterns in text, regular expressions can be extended with capturing groups. Capturing groups are used to extract information from text, which can be used independently from the originally matched text.

FOREST, after synthesising a regular expression to describe the desired format for a certain input field, completes it with capturing groups that, either capture some information desired by the user, or further restrict the accepted values by enforcing integer conditions over the integer values captured in the input string. In this chapter we describe the synthesis procedure that generates the second and third components of the regex validation: capturing groups that reflect the captures provided with the valid examples and capture conditions which express integer conditions for the values in the example that are satisfied by all valid examples but not by any of the conditional invalid examples.

In Section 5.1, we look at the base enumeration cycle that produces capturing groups over a given regular expression. This enumeration method is used both in the synthesis of capturing groups and capture conditions. Then, in Section 5.2, we look at how FOREST synthesises the second component in the regex validations. After, we move on to the synthesis of the third component in the regex validations: capture conditions. In Section 5.3, we explain the SMT encoding used to produce a minimum set of integer conditions over some capturing groups. Finally, in Section 5.4, we show how we can produce a distinguishing input to resolve the ambiguity of the input examples when there are several minimum sets of conditions that satisfy them.

5.1 Enumeration

The synthesis of both the capturing groups and the capture conditions is done using enumerative search. Thus, the first step in both synthesis procedures is to enumerate capturing groups over the previously computed regular expression.

To enumerate capturing groups, FOREST starts by identifying atomic sub-regexes in the regular expression. Atomic regexes correspond to the smallest sub-regexes whose concatenation results in the original complete regex. It does not make sense to place a capturing group inside atomic sub-regexes.

For example, the regex `[0-9]{2}` is an atomic sub-regex: there are no smaller sub-regexes whose concatenation results in it. It does not make sense to place a capturing group on just a part of it: `([0-9]){2}` does not have a clear meaning. Once identified, the atomic sub-regexes are placed in an ordered list. The concatenation of all elements in this list results in the original regular expression.

Enumerating capturing groups over the regular expression is done by enumerating non-empty disjoint sub-lists of this list. The elements inside each sub-list form a capturing group.

Example 5.1.1. Recall the previously shown date regex: `[0-9]{2}/[0-9]{2}/[0-9]{4}`. The ordered list of atomic sub-regexes for this regex is [`[0-9]{2}`, `/`, `[0-9]{2}`, `/`, `[0-9]{4}`].

Suppose we want to get a single capturing group within the date regex. A single capturing group is a single sub-list of the list of atomic sub-regexes. The following are some examples of sub-lists of the atomic sub-regexes list and their resulting capturing groups:

[[`[0-9]{2}`], `/`, `[0-9]{2}`, `/`, `[0-9]{4}`] → `([0-9]{2})/[0-9]{2}/[0-9]{4}`

[[`[0-9]{2}`, `/`], `[0-9]{2}`, `/`, `[0-9]{4}`] → `([0-9]{2}/)[0-9]{2}/[0-9]{4}`

[`[0-9]{2}`, `/`, [`[0-9]{2}`, `/`, `[0-9]{4}`]] → `[0-9]{2}/([0-9]{2}/[0-9]{4})`

Analogously, to enumerate two or more capturing groups, we consider two or more non-empty disjoint sub-lists. For the date regex, we can have at most 5 capturing groups because the atomic sub-regexes list has 5 elements. The following are some examples of multiple sub-lists of the atomic sub-regexes list and their resulting capturing groups:

[[`[0-9]{2}`], `/`, [`[0-9]{2}`], `/`, `[0-9]{4}`] → `([0-9]{2})/([0-9]{2})/[0-9]{4}`

[[`[0-9]{2}`], `/`, [`[0-9]{2}`, `/`, `[0-9]{4}`]] → `([0-9]{2})/([0-9]{2}/[0-9]{4})`

[[`[0-9]{2}`], `/`, [`[0-9]{2}`], `/`, [`[0-9]{4}`]] → `([0-9]{2})/([0-9]{2})/([0-9]{4})`

5.2 Groups Synthesis

The second component of the regex validation are capturing groups that reflect the captures provided with the valid examples. To synthesise these capturing groups, FOREST enumerates capturing groups over the produced regular expression as explained in Section 5.1. Since the captured values are provided along with all valid examples, it is known how many capturing groups are required.

For each enumerated set of capturing groups, FOREST matches the regular expression with the capturing groups to each of the valid example strings. The enumeration process continues until a set of capturing groups that produces the user-provided captures is found. Once the correct captures are found, they are stored as part of the solution.

Example 5.2.1. Recall the motivating example in Section 1.1: the user wished not only to validate the input string but also to extract some information from it. The goal was to extract the year from each date, so it could be used afterwards. In this situation, the examples provided by the user are:

19/08/1996, 1996	22/09/2000, 2000	29/09/2003, 2003
26/10/1998, 1998	01/12/2001, 2001	31/08/2015, 2015

As we saw back then, FOREST can provide capturing groups within the regular expression that allow the user to achieve this. In this case, each example is followed only by one capture, so we want to enumerate only one sub-list from the list of atomic regexes. The desired sub-list and corresponding capturing group are:

`[[0-9]{2}, /, [0-9]{2}, /, [[0-9]{4}]]` $\rightarrow$ `[0-9]{2}/[0-9]{2}/([0-9]{4})`

5.3 Conditions Synthesis

The third component of the regex validation is a set of integer conditions over captured values in the example. The conditions are satisfied by all valid examples but not by any of the conditional invalid examples.

At this point, FOREST has a regular expression that matches all valid examples. In order to compute the captures using the regular expression, we need all conditional invalid examples to be matched by this regular expression as well. To ensure this, we remove from the set of conditional invalid examples all those that are not matched by the regular expression. This is equivalent to moving them from the conditional invalid set to the invalid set. To the user, it matters only that all invalid and conditional invalid examples are rejected by the final regex validation; whether they are rejected because they do not match the regular expression or because they do not satisfy the capture conditions is not relevant. Therefore, this behaviour is inconsequential to the user.

To synthesise the capture conditions, FOREST starts by enumerating capturing groups using the enumeration process described in Section 5.1. When synthesising capture conditions, the number of necessary capturing groups is not known beforehand. Thus, we enumerate capturing groups in progressively increasing number. We start by attempting to generate conditions with just one capturing group; if no solutions are found, we enumerate two capturing groups, and so on.

Since it is our goal to synthesise integer conditions, we require that the captures resulting from any given set of capturing groups all correspond to integers. This is done by, for each enumerated capturing groups, matching the regex to all valid and conditional invalid examples and casting the resulting captures to integers. If this is not successful for all captures and all examples, these capturing groups cannot be used for capture conditions. The current capturing groups are discarded and the enumeration process continues.

For each enumerated set of integer capturing groups, FOREST tries to find a set of capture conditions over those capturing groups that are satisfied by all valid examples but not by any of the conditional

invalid examples. A capture condition is a 3-tuple: it contains the captured text, an integer comparison operator and an integer argument. In this document, we use the notation $\$i, i \in 0, 1, ...$, to reference the text captured by the $(i+1)^{\text{th}}$ group. In FOREST, we consider only two integer comparison operators, $\leq$ and $\geq$. However, the algorithm can be easily extended to include more integer comparison operators. We do not allow two capture conditions referring to the same capturing group that use the same operator, changing only the integer argument. Therefore, a capture condition can be identified by the capturing group's index and the integer comparison operator.

Let $\mathcal{C}$ be a set of capturing groups and $\mathcal{C}(x)$ the integer captures that result from applying $\mathcal{C}$ to example string x. Let $\mathcal{O}$ be the set of integer operators being considered; in FOREST $\mathcal{O} = \{\leq, \geq\}$. Then, let $\mathcal{D}_\mathcal{C}$ be the set of all possible capture conditions over capturing groups $\mathcal{C}$. $\mathcal{D}_\mathcal{C}$ results from combining each capturing group with each integer operator: $\mathcal{D}_\mathcal{C} = \mathcal{C} \times \mathcal{O}$. Finally, let $\mathcal{V}$ be the set of all valid examples, $\mathcal{I}$ the set of all conditional invalid examples and $\mathcal{X} = \mathcal{V} \cup \mathcal{I}$ the set of all examples. Given a set of capturing groups $\mathcal{C}$, FOREST uses MaxSMT to select from $\mathcal{D}_\mathcal{C}$ the minimum set of capture conditions that are satisfied by all valid examples and by none of the conditional invalid. To encode the problem, we use the following sets of Boolean variables:

- a_x for every example $x \in \mathcal{X}$. $a_x =$ *True* means that example x satisfies all capture conditions in the solution.
- $s_{cap,x}$ for every capture *cap* $\in \mathcal{C}(x)$ and examples $x \in \mathcal{X}$. $s_{cap,x} =$ *True* means that capture *cap* in example x satisfies all used conditions that refer to it.
- u_{cond} for all conditions *cond* $\in \mathcal{D}_\mathcal{C}$. $u_{cond} =$ *True* means condition *cond* is used in the solution.

Additionally, we define a set of integer variables b_{cond}, for all conditions *cond* $\in \mathcal{D}_\mathcal{C}$ that represent the integer argument in each condition.

Then, we define a set of constraints that ensure the set of selected captures correctly classifies the examples. Let SMT(*cond*, x) be the SMT representation of the condition *cond* for example x. In this representation, the capture is an integer value, the integer operator is has the usual semantics, and the integer argument is the corresponding b_{cond} integer variable.

Let $\mathcal{D}_{cap} \subseteq \mathcal{D}_\mathcal{C}$ be the set of capture conditions that refer to capture *cap*. Constraint 5.1 states that a capture *cap* in example x satisfies all conditions if and only if for every condition that refers to *cap*, either it is not used or it is satisfied by the value of that capture in that example.

$$s_{cap,x} \leftrightarrow \bigwedge_{cond \in \mathcal{D}_{cap}} u_{cond} \rightarrow \text{SMT}(cond, x) \tag{5.1}$$

Example 5.3.1. Recall the first valid example from the motivating example in Section 1.1: x_0 = "19/08/1996". Suppose FOREST has already synthesised the desired regular expression and enumerated the capturing group that corresponds to the day: `([0-9]{2})/[0-9]{2}/[0-9]{4}`. Let $cond_0$ and $cond_1$ be the conditions that refers to the first (and only) capturing group, $\$0$, and operators $\leq$ and $\geq$

respectively. The SMT representation for $cond_0$ and x_0 is

$$\text{SMT}(cond_0, x_0) \;=\; 19 \leq b_{cond_0}$$

.

Constraint 5.1 for this example is then:

$$s_{0,x_0} \leftrightarrow (u_{cond_0} \rightarrow (19 \leq b_{cond_0})) \wedge (u_{cond_1} \rightarrow (19 \geq b_{cond_1})).$$

Constraint 5.2 states that an example x satisfies all capture conditions that refer to it if and only if all the individual captures satisfy their respective conditions.

$$a_x \leftrightarrow \bigwedge_{cap \in \mathcal{C}(x)} s_{cap,x} \tag{5.2}$$

Finally, constraint 5.3 ensures that used conditions are satisfied by all valid examples and none of the conditional invalid examples.

$$\bigwedge_{x \in \mathcal{V}} a_x \;\wedge\; \bigwedge_{x \in \mathcal{I}} \neg a_x \tag{5.3}$$

Since we are looking for the minimum set of capture conditions, we add soft clauses to penalise the usage of each capture condition in the solution:

$$\bigwedge_{cond \in \mathcal{D}_\mathcal{C}} \neg\, u_{cond}. \tag{5.4}$$

Running a SMT solver with these constraints results in a solution. We consider part of the solution only the capture conditions whose u_cond is *True* in the resulting model. We also extract the values of the integer arguments in each condition from the model values of the b_{cond} variables.

5.4 Conditions Disambiguation

The same problem we encountered while synthesising a regular expression emerges for capture conditions: the ambiguity of the examples as a specification. There can be many sets of capture conditions that validate all valid examples and invalidate all condition invalid. To ensure the solution meets the user's needs, FOREST disambiguates the specification using a procedure based on distinguishing inputs similar to that used during the synthesis of the regular expression.

First, FOREST uses the SMT solver to produce another valid solution. This is done, like before, by adding a clause that blocks the current model and then asking the solver to solve the new formula. To block the current model, let $\mathcal{D}^+$ be the set of conditions that were part of the solution in the current model, i.e., the set of conditions *cond* for which u_{cond} was assigned *True*. Let $m(x)$ represent the value of variable x in the current model. To get a different solution, we add constraint 5.5 to ensure that the

integer value is different in at least one of the selected captures.

$$\bigvee_{cond \in \mathcal{D}^+} b_{cond} \neq m(b_{cond}) \tag{5.5}$$

Another call to the SMT solver supplies another, different solution. We have then $\mathcal{S}_1$ and $\mathcal{S}_2$, two different solutions, i.e., two different sets of capture conditions that satisfy the specification. Our goal is to find a distinguishing input: a string c which is satisfies all capture conditions in $\mathcal{S}_1$, but not those in $\mathcal{S}_2$, or vice-versa. First, to simplify the problem, FOREST eliminates from $\mathcal{S}_1$ and $\mathcal{S}_2$ conditions which are present in both: these are not relevant to compute a distinguishing input. Let $\mathcal{S}_1^*$ (resp. $\mathcal{S}_2^*$) be the subset of $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) containing only the distinguishing conditions, i.e., the conditions that differ from those in $\mathcal{S}_2$ (resp. $\mathcal{S}_1$).

Example 5.4.1. Recall the valid and conditional invalid examples from the motivating example in Section 1.1:
Valid examples:

19/08/1996	22/09/2000	29/09/2003
26/10/1998	01/12/2001	31/08/2015

Conditional invalid examples:

33/08/1996	22/13/2000	12/31/2003
26/00/1998	00/12/2001	52/03/2015

Because there is no conditional invalid example that shows that there cannot be a date with the day 32, FOREST can produce two correct sets of capture conditions for the examples provided in over `([0-9]{2})/([0-9]{2})/[0-9]{4}`:

- $\mathcal{S}_1 = \$0 \leq 31 \wedge \$0 \geq 1 \wedge \$1 \leq 12 \wedge \$1 \geq 1$
- $\mathcal{S}_2 = \$0 \leq 32 \wedge \$0 \geq 1 \wedge \$1 \leq 12 \wedge \$1 \geq 1$

To find the correct solution, FOREST first keeps only the captures that differ from one solution to the other: $\mathcal{S}_1^* = \$0 \leq 31$, and $\mathcal{S}_2^* = \$0 \leq 32$.

Recall that $\mathcal{S}_1$ and $\mathcal{S}_2$ are capture conditions with respect to the same set of capturing groups $\mathcal{C}$. The capture conditions do not refer to the whole matched string, but only to parts of it (the captured portions). Thus, we do not compute the distinguishing string c directly. Instead, we compute the integer value of the distinguishing captures in c: the captures that result from applying the regular expression and its capturing groups to the distinguishing input string. We define $|\mathcal{C}|$ integer variables, c_i, which correspond to the values of the distinguishing captures:

$$c_0, c_1, ..., c_{|\mathcal{C}|} = \mathcal{C}(c).$$

Example 5.4.2. In Example 5.4.1 we had two solutions: $\mathcal{S}_1^* = \$0 \leq 31$ and $\mathcal{S}_2^* = \$0 \leq 32$. Even though the regular expression had two capturing groups, since only one is present in the distinguishing conditions, we can define only one integer variable, c_0, representing the integer value of the first capturing group in the distinguishing string.

We define SMT(*cond*, c), the SMT representation of each condition *cond*, for the distinguishing string using the c_i. Each capture in $\mathcal{C}(c)$ is represented by its respective c_i variable, the operator maintains it usual semantics and the integer argument is its value in the solution to which the condition belongs.

Finally, we add constraint (5.6), which states that c satisfies the conditions in one solution but not the other.

$$\bigwedge_{cond \in \mathcal{S}_1^*} \text{SMT}(cond, c) \quad \neq \bigwedge_{cond \in \mathcal{S}_2^*} \text{SMT}(cond, c). \tag{5.6}$$

Example 5.4.3. Recall two solutions from Example 5.4.1, $\mathcal{S}_1^* = \$0 \leq 31$ and $\mathcal{S}_2^* = \$0 \leq 32$. To find c_0, the value of the distinguishing capture for these solutions, we solve the constraint

$$\exists c_0 : c_0 \leq 31 \neq c_0 \leq 32$$

and get the value $c_0 = 32$ which satisfies $\mathcal{S}_2^*$ (and $\mathcal{S}_2$), but not $\mathcal{S}_1^*$ (or $\mathcal{S}_1$).

If we pick the first valid example, “19/08/1996” as basis for c, the respective distinguishing input is c = “32/08/1996”. Once the user classifies c as invalid, c is added as a conditional invalid example and $\mathcal{S}_2$ is removed from consideration.

Note that string c is not a string variable: it is not part of the encoding and it is invisible to the solver. We include it here as an auxiliary structure. Therefore, the model that results from a call to the SMT solver does not give us a distinguishing string c. Instead, it outputs only values for the distinguishing captures c_i. To produce the distinguishing string c, FOREST picks an example from the valid set, applies to it the regular expression with the capturing groups, and replaces its captures with the model values for c_i. An additional step is required here to pad with zeros c_i's integer values so that their string representation has the same length as the captures they are replacing in the valid string.

Chapter 6

Experimental Results

Implementation. FOREST is implemented in Python 3.8 on top of TRINITY, a general-purpose synthesis framework [39]. All SMT formulas are solved using the Z3 SMT solver, version 4.8.9 [12]. To find distinguishing inputs in regular expression synthesis, FOREST uses Z3's theory of regular expressions (part of the theory of strings [6]). To check the enumerated regexes against the examples, we use Python's regex library [52]. The results presented herein were obtained using an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, with 64GB of RAM, running Debian GNU/Linux 10. All processes were run with a time limit of one hour.

Benchmarks. To evaluate FOREST, we used 69 benchmarks based on real-world form-validation regular expressions. These were collected from regular expression validators in validation frameworks and from `regexlib` [57], a website where users can upload their own regexes. Among these 69 benchmarks there are different formats: national IDs, identifiers of products, date and time, vehicle registration numbers, postal codes, email and phone numbers. For each benchmark, we randomly generated sets of strings to be used as input examples for FOREST. All 69 benchmarks require a regular expression to validate the examples. 10 require capturing groups to extract the captures provided alongside the valid examples, and 8 require additional capture conditions to validate conditional invalid examples. On average, each instance is composed of 13.4 valid examples (ranging from 4 to 33) and 9.0 invalid (ranging from 2 to 38). The 7 instances that target capture conditions have on average 6 conditional invalid examples (ranging from 4 to 8). The median number of total examples per instance is 22, while the median number of valid examples is 11, 7 for invalid, and 6 for conditional invalid.

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

Q1. How does FOREST compare against REGEL? (Section 6.1)

Q2. How does pruning affect multi-tree's time performance? (Section 6.2)

Q3. How does static multi-tree improve on dynamic multi-tree? (Section 6.2)

Q4. How does multi-tree compare against other enumeration-based encodings? (Section 6.3)

Table 6.1: Comparison of time performance using different synthesis methods.

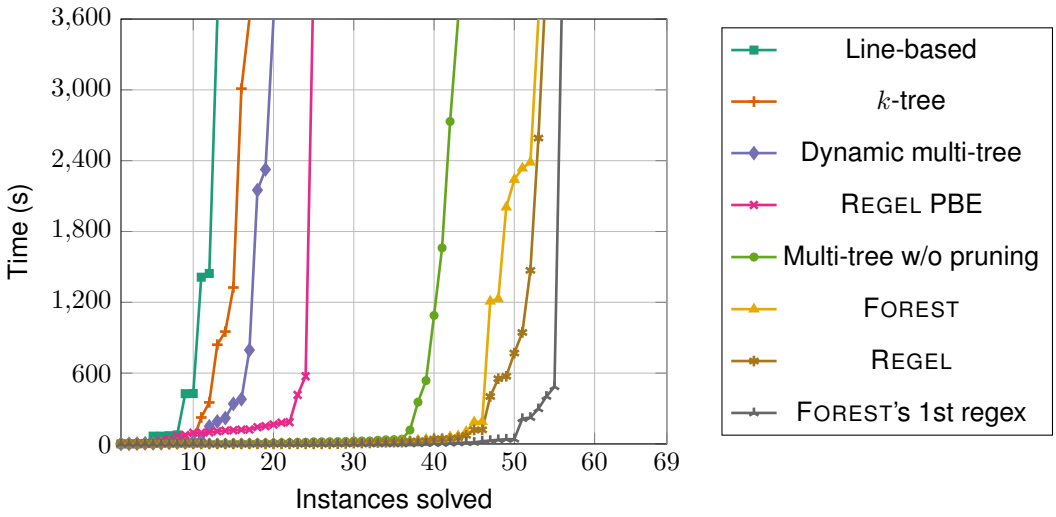
Timeout (s)	**10**	**60**	**3600**
FOREST (w/ interaction)	33	42	52
FOREST's 1st regex (no interaction)	44	51	55
FOREST w/ multi-distinguish interaction	28	36	46
Multi-tree w/o pruning	21	36	42
Dynamic-only multi-tree	6	11	19
k-tree	4	10	15
Line-based (w/o pruning)	4	4	12
REGEL	32	43	53
REGEL PBE	7	9	29

Q5. How many examples are required to return an accurate solution? (Section 6.4)

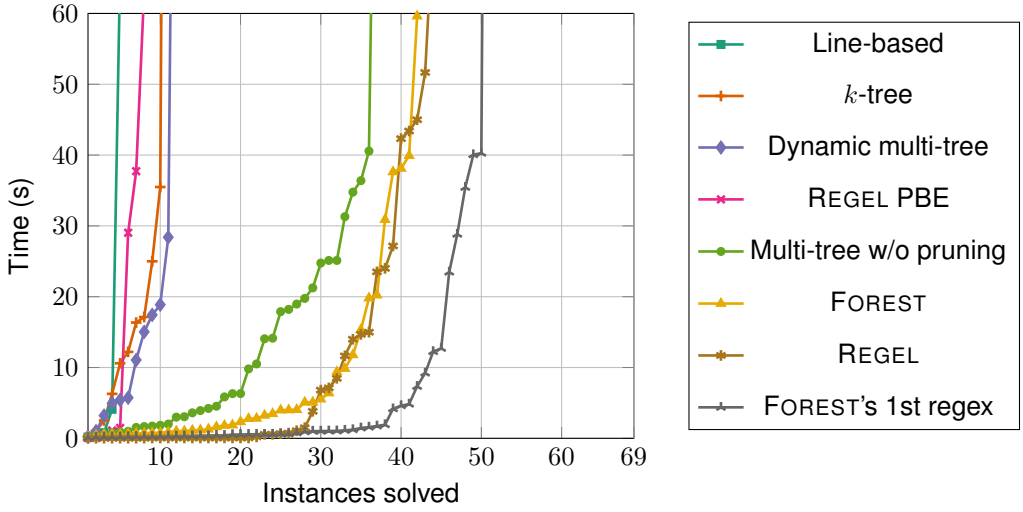
Q6. How does sketching and a multi-distinguish interaction affect FOREST's time performance? (Section 6.5)

Table 6.1 shows the number of instances synthesised in under 10, 60 and 3600 seconds using FOREST, as well as using the different variations of the synthesizer which will be described in the following sections. The first row shows the synthesis time using FOREST's default configuration. FOREST, by default, uses static multi-tree when it is possible to split the examples with dividing substrings, and dynamic multi-tree when it is not. In practice, static multi-tree is used in 57 of the 69 benchmarks (83%). The default setting has all pruning techniques enabled, does not use sketch-based enumeration, and disambiguates the examples using conversational clarification. The second row in the table refers to FOREST's synthesis times with no interaction, in which case the returned solution is simply the first one found. In the third row, we show the synthesis time using the multi-distinguish interaction model. The following two rows show FOREST's time performance with variations to the enumeration procedure: with pruning disabled and applying dynamic multi-tree enumeration in all benchmarks. Rows 6 and 7 show synthesis times using different encodings for regex enumeration, k-tree and line-based. Finally, we also compare FOREST's synthesis times with REGEL [8]. REGEL takes as input a natural description of user intent, used to generate a sketch, as well as examples, which are used to complete the sketch. We show the sketch-completion time of the complete REGEL synthesizer, and synthesis time of the PBE engine of REGEL by itself, which we denote by REGEL PBE. When using the complete REGEL, the synthesis time includes not only sketch-completion times (indicated in the table), but also sketch-generation, which takes on average 60 seconds per benchmark.

The cactus plots in Figure 6.1 show the cumulative synthesis time on the y-axis plotted against the number of benchmarks synthesised by each variation of FOREST (on the x-axis). In both plots in Figure 6.1, each line corresponds to a row in Table 6.1. The synthesis methods that correspond to lines more to the right of the plot are able to find a solution to more benchmarks in less time. Figure 6.1a shows



(a) Instances solved in 3600 seconds.



(b) Instances solved in 60 seconds.

Figure 6.1: Comparison of number of instances solved using different methods.

the benchmarks synthesised before the 3600 seconds timeout and Figure 6.1b the number solved in the first 60 seconds.

FOREST correctly finds a solution 33 benchmarks (48%) in under 10 seconds. In one hour, FOREST synthesises 52 benchmarks (75%), with 98% accuracy: only one solution did not correspond to the desired regex validation. This happens because FOREST disambiguates only among programs at the same depth. In this instance, the first solution is at a lower depth as the correct one, thus the correct solution is never found. After 1 hour of running time, FOREST is interrupted, but it prints its current best validation before terminating. After the timeout, FOREST returned 2 more regexes, both the correct solution for the benchmark. FOREST never uses more than 500MB of RAM in each benchmark. In all benchmarks to which FOREST returns a solution, the first matching regular expression is found in under 10 minutes. In 44 benchmarks, the first regex is found in under 10 seconds. The remaining time is spent looking for better answers and disambiguating the input examples. FOREST interacts with the user to

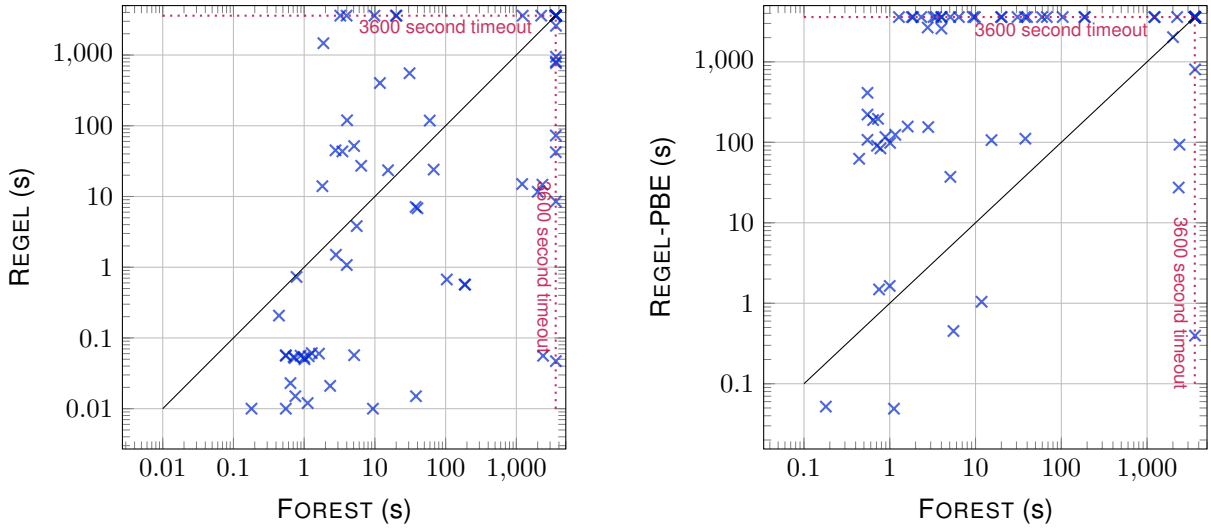
disambiguate the examples in 29 benchmarks. Overall, it asks 1.7 questions and spends 37.2 seconds computing distinguishing inputs, on average. Without interaction, FOREST finds a solution a lot faster. 44 regexes are synthesised in the first ten seconds, 55 after one hour. However, the accuracy is much lower: instead of 51 correct validations, we have only 34 correct validations. This is a severe decrease in accuracy, from 98% to 62%.

FOREST is able to synthesise the capturing groups that match the captures provided with each valid example in 9 out of 10 instances. In the 9 instances in which capturing groups are computed successfully, the capturing groups are computed in less than half a second. In the remaining instance, the first regex FOREST finds that satisfies all examples is not compatible with the required captures. In this particular situation, the regex should identify HTML-format colours, which consist in a '#' followed by 6 hexadecimal digits, 3 groups of 2 digits, each referring to a colour component, e.g. '#0A1B3C'. Furthermore, the capturing groups should extract the values corresponding to each colour component. For the same example, the desired captures are ('0A','1B','3C'). To validate the format, FOREST synthesises the regex `#[0-9A-F]{6}` in under 1 second. However, there are no capturing groups over this regex that produce the desired captures. To produce the captures, we must first synthesise the regex `#([0-9A-F]{2})([0-9A-F]{2})([0-9A-F]{2})`, and FOREST does not enumerate that regex within the time limit (1 hour). Therefore, this instance times out.

No instances time out during the synthesis of capture conditions. In 6 of the benchmarks, we need only 2 capturing groups and at most 4 conditions. In these 6 instances, the conditions' synthesis takes under 2 seconds. The remaining 2 benchmarks need 4 capturing groups and take longer. It takes 99 seconds to synthesise 4 capturing groups with 4 conditions. The instance that takes the longest to compute capture conditions is quite complex. The synthesised regular expression has 23 nodes, and it requires 6 capture conditions over 4 capturing groups. The result validates a datetime format: `[0-9]{4}/([0-9]{2})/([0-9]{2}) ([0-9]{2}):([0-9]{2}) AM|PM`, $\$0 \leq 1 \wedge \$0 \leq 12 \wedge \$1 \leq 1 \wedge \$1 \leq 31 \wedge \$2 \leq 12 \wedge \$3 \leq 59$. This benchmark takes 1229 seconds (about 20 minutes) to synthesise, 97% of which is spent synthesising the capture conditions. During capture conditions synthesis, FOREST interacts 6.75 times and takes 0.1 seconds to compute distinguishing inputs, on average.

6.1 Comparison with REGEL

As mentioned in Section 3.5, REGEL's synthesis procedure is split into two steps: sketch generation (using a natural language description of desired behaviour) and sketch completion (using input-output examples). To compare REGEL and FOREST, we extended our 69 form validation benchmarks with a natural language description. To assess the importance of the natural language description, we also ran REGEL using only its PBE engine. In Figure 6.2a we can compare FOREST's total synthesis time with REGEL's sketch completion time. Each mark in the plot represents an instance. The value on the y-axis shows REGEL's sketch completion time and the corresponding value on the x-axis is FOREST's synthesis time for the same instance. The marks above the $y = x$ line (also represented in the plot) represent problems that took longer to synthesise with REGEL than with FOREST. Figure 6.2b shows FOREST's



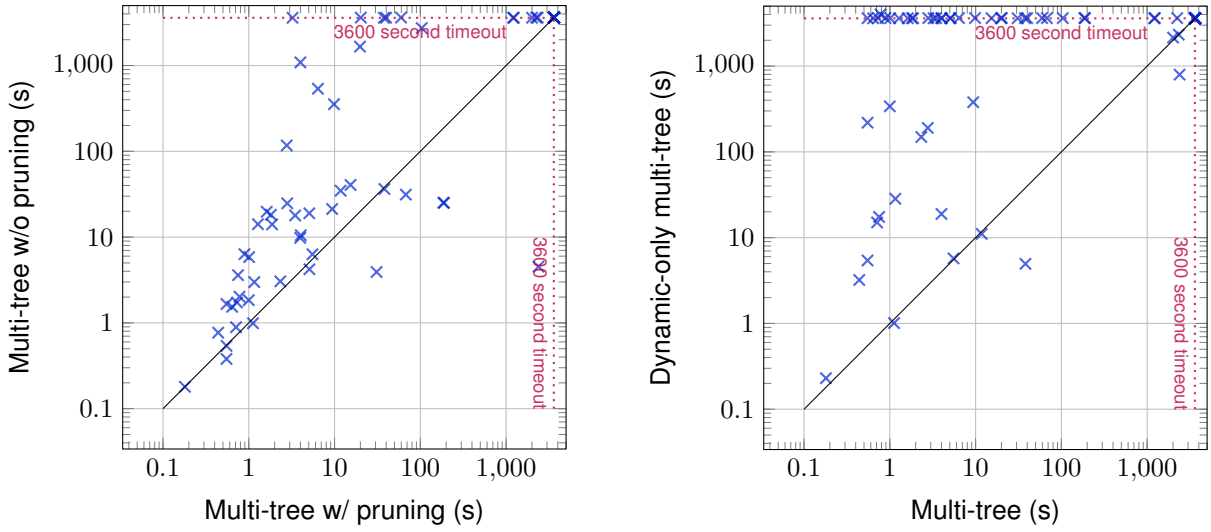
(a) FOREST vs. REGEL

(b) FOREST vs. REGEL-PBE

Figure 6.2: Comparison of synthesis time of REGEL and FOREST

total synthesis time against REGEL-PBE's synthesis time. Sketch generation took on average 60 seconds per instance, and successfully generated a sketch for 68 instances. The remaining instance was run without a sketch. We considered only the highest ranked sketch for each instance. In Table 6.1 we show how many instances can be solved with different time limits for sketch completion; note that these values do not include the sketch generation time. REGEL returned a regular expression for 53 instances and timed out after one hour in the remaining 16. Since REGEL does not implement a disambiguation procedure, the returned regular expression does not always exhibit the desired behaviour, even though it correctly classifies all examples. Of the 53 synthesised expressions, 35 exhibit the desired intent (only one more than FOREST's first solution). This is a 66% accuracy, which is much lower than FOREST's at 98%. We conclude that, in terms of synthesis time, REGEL is comparable to FOREST, which solves only one fewer instance. However, REGEL's accuracy for the desired validation is much lower than that of FOREST, as they don't implement an interaction model. In terms of accuracy, REGEL is comparable to FOREST's 1st solution, which has lower synthesis times in general.

55 out of the 68 generated sketches are of the form $\square\{S_1, ..., S_n\}$, where each S_i is a concrete sub-regex, i.e., has no holes. This construct indicates the desired regex must contain *at least* one of $S_1, ..., S_n$, and contains no information about the top-level operators that are used to connect them. 24 of the 53 synthesised regexes are based on sketches of that form, and they result from the direct concatenation of *all* components in the sketch. No new components are generated during sketch completion. Thus, most of REGEL's sketches could be integrated into FOREST, whose multi-tree structure holds precisely those top-level operators that were missing from REGEL's sketches. As can be seen in Figure 6.2b and Table 6.1, REGEL's performance is severely impaired when using only its PBE engine. We predict integrating REGEL's sketches with FOREST would have a positive outcome, since FOREST shows better performance working only from examples.



(a) Impact of pruning

(b) Impact of example-splitting

Figure 6.3: Comparison of synthesis time using different variations of FOREST's multi-trees.

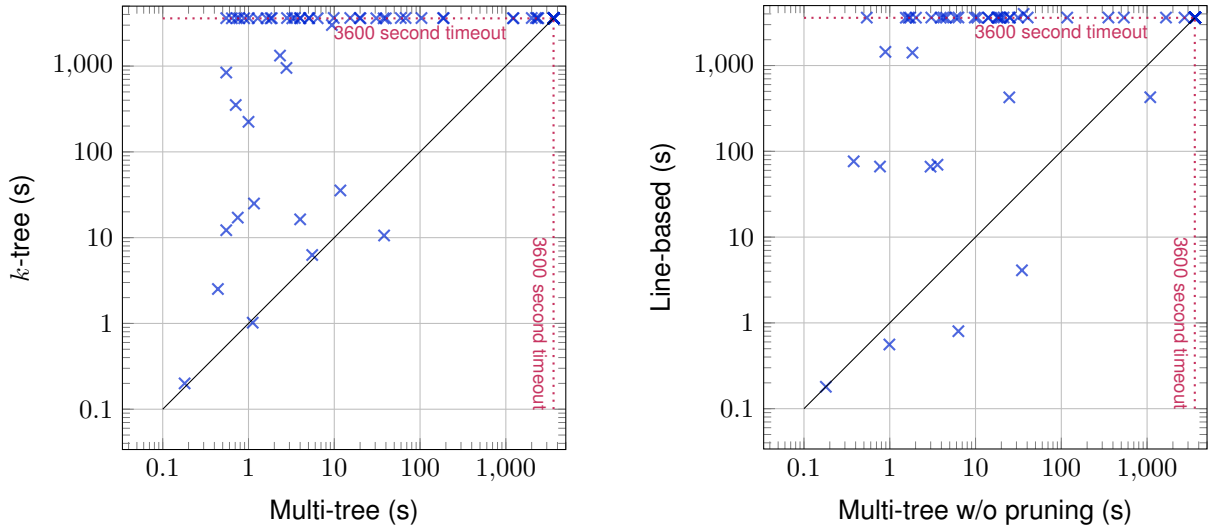
6.2 Pruning the Search Space and Splitting Examples

To evaluate the impact of pruning the search space as described in Section 4.2.3, we ran FOREST with all pruning techniques disabled. In the scatter plot in Figure 6.3a, we can compare the synthesis time on each benchmark with and without pruning. On average, with pruning, FOREST can synthesise regexes in half the time and enumerates less than half of the regexes before returning. There is no significant change in the number of interactions before returning the desired solution.

FOREST is able to split the examples and use static multi-tree as described in Section 4.2.2 in 57 benchmarks (83%). The remaining 12 are solved using dynamic multi-tree. To assess the impact of using static multi-tree we ran FOREST with a version of the multi-tree enumerator that does not attempt to split the examples, and jumps directly to dynamic multi-tree enumeration. In the scatter plot in Figure 6.3b, we compare the synthesis times of each benchmark. Using static multi-tree when possible, FOREST requires, on average, about two thirds of the time (68%) to return the desired regex for the benchmarks solved by both methods. Furthermore, static multi-tree allows FOREST to synthesise more complex expressions: the maximum number of nodes in a solution returned by dynamic multi-tree is 12 (average 6.7), while complete multi-tree synthesises expressions of up to 24 nodes (average 10.2).

6.3 Multi-tree versus *k*-tree and Line-based Encodings

To evaluate the performance of the multi-tree encoding, we ran FOREST with two other enumeration encodings: k-tree and line-based. The latter is a state of the art encoding for the synthesis of SQL queries [45, 47]. k-tree is implemented as the default enumerator in TRINITY [39], while the line-based enumerator is used as available in SQUARES [46]. The k-tree encoding (introduced in Section 4.2.1) has a very similar structure to that of multi-tree, so our pruning techniques were easily applied to this encoding. On the other hand, line-based encoding is intrinsically different, making the pruning techniques harder



(a) Multi-tree vs. k-tree

(b) Multi-tree vs. line-based

Figure 6.4: Comparison of synthesis time using different encodings.

to implement. Because of this, we compare line-based encoding to multi-tree without pruning. In every other aspect, the three encodings were run in the same conditions, using FOREST's regex DSL. The synthesis time comparison of multi-tree with k-tree and line-based encodings are shown in the scatter plots in Figure 6.4a and Figure 6.4b, respectively. k-tree is able to synthesise programs with up to 10 nodes, while the line-based encoding synthesises programs of up to 9 nodes. Neither encoding outperforms multi-tree.

As can be seen in Table 6.1 and Figure 6.1, line-based encoding does not outperform the tree-based encodings in our experiments. This contradicts the results obtained by Orvalho et al. [45] for the synthesis of SQL queries. We conjecture the reason behind this disparity arises from different nature of the DSLs. Most SQL queries, when represented as a tree, leave many branches of the tree unused, which results in the need of a larger depth for the tree. Regular expressions, on the other hand, take more advantage of the tree structure.

6.4 Fewer Examples

To assess the impact of providing fewer examples on the accuracy of the solution, we ran FOREST with modified versions of each benchmark. First, each benchmark was run with at most 10 valid and 10 invalid examples, chosen randomly among all examples. Conditional invalid examples are already very few per instance, so these were not altered. The accuracy of the returned regexes is slightly lower.

With only 10 valid and 10 invalid examples, FOREST returns the correct regex in 96% of the benchmarks, which represents a very slight decrease compared with results with all examples, where the accuracy was 98%. We also saw an increase in the number of interactions before returning, since fewer examples are likely to be more ambiguous. With only 10 examples, FOREST interacts on average 2.1 times per instance during regex synthesis, which represents an increase of about a fourth compared to FOREST run with all examples, which interacted on average 1.7 times per instance.

After, we reduced the number of examples even further: only 5 valid and 5 invalid. The accuracy of FOREST in this setting was reduced to 71%. On average, it interacted 4.0 times per benchmark, which is over two times more than before.

We conclude that FOREST maintains a very high accuracy with 10 examples of each kind, but not with 5 examples of each kind. Therefore, to ensure the solution is correct, the user should provide FOREST at least 10 examples of each kind.

6.5 Sketching and Multi-distinguish Interaction

We implemented sketch-based enumeration in FOREST with both sketch-completion methods described in Section 4.2.4: graph-based and SMT-based. Graph-based sketch completion solved only 4 instances, all in under 1 second. All 4 successfully synthesised instances are among the simplest in the benchmark set, the synthesised regex has only 4 nodes, and the correct sketch was enumerated within the first 6 sketches. This means FOREST had to exhaust very few sketches (at most 5) before finding the correct one. In sum, the only instances that are solved with graph-based sketching are those where the regex is simplest and whose correct sketches happens to be enumerated first. In the remaining instances, FOREST enumerates and tests at most 500 sketches before timeout, while sketch-less FOREST enumerates at most 30 thousand regexes before the timeout. Graph-based sketch completion is inefficient because the possible values for each leaf are not pruned, and all combinations are tested.

SMT-based sketching performed only slightly better: 8 instances were solved in total. Synthesis times vary in accordance with each solution's complexity. Four 4-node regexes synthesised in under 2 seconds, one 5- and one 6-node regex synthesised in under one minute, and the remaining two 8-node solutions in under one hour. Of the unsolved instances, only one timed out. The remaining 47 were forcefully terminated due to insufficient memory within the imposed 4GB limit. We observed out-of-memory errors even with the limit set as high as 32GB. This happens because the SMT formula grows exponentially with the number of holes in the sketch, so larger sketches quickly become intractable.

We also compared the behaviour of the two interaction models described for regex synthesis in Section 4.3. The multi-distinguish model was run every 4 regexes found, because our experiments showed that Z3's regex theory could not efficiently distinguish more than 4 regexes at a time. Regardless of the interaction model, FOREST interacts with the user in 29 benchmarks. With multi-distinguish, of these 29, only 23 successfully terminate synthesis and compute a regex. One of the failed instances times out before enumerating the correct regex. The remaining 5 timeout after finding 4 regexes in the first minute of synthesis and spending the remaining 59 minutes unsuccessfully trying to find a distinguishing input for them. While 4 instances we distinguishing relatively complex regexes (10-16 nodes), the remaining one times out trying to distinguish simple 4-node regular expressions.

Regarding the number of interactions, 13 instances required 3 or more interactions using the multi-distinguish model, which is 3 fewer than with conversational clarification. With multi-distinguish, FOREST required at most 9 regex interactions before converging to a solution. With conversational clarification, FOREST interacts up to 21 times to synthesise the correct regular expression. Overall, considering only

benchmarks synthesised in both runs, with conversational clarification FOREST spends about half the time (3.2 seconds on average) computing distinguishing input than with multi-distinguish (6.2 seconds on average).

As expected, when maximising the number of programs eliminated by each interactions, FOREST had on average fewer interactions with the user. Although it successfully minimises the number of required interactions to completely disambiguate the specification, we conclude that multi-distinguish is not a viable alternative to conversational clarification for two motives: (i) it takes a lot longer to compute a distinguishing input, even with as few as 4 programs to distinguish, and (ii) Z3's regex theory is unpredictable: it efficiently distinguishes some complex regexes but also fails with simple ones. Thus, FOREST uses conversational clarification to interact with the user by default.

Chapter 7

Conclusions and Future Work

As computer applications become increasingly dependent on the collection and subsequent analysis of large amounts of data, the need arises to make these data collection and analysis tasks available to users who do not possess a programming background. Digital forms are often used for structured data collection. Part of what makes them desirable is the ability to enforce a certain format on the inputs, by adding real-time validations to the input fields. Validations help prevent 'typos' and format inconsistencies, which leads to clean and standardised data. Regular expressions are a very expressive and commonly used method to enforce patterns and validate the input fields of digital forms. However, writing regex validations requires specialised knowledge that not all users possess. Furthermore, when done for a large number of input fields, the task of hand-writing regexes becomes monotonous and error prone. To help users write regular expressions, previous work has focused on the synthesis of regexes from examples and from natural language. However, to the best of our knowledge, none of these methods for regex synthesis has form validations as the main focus.

In this dissertation, we have presented a new algorithm for synthesis of regex validations from a set example values for the input field. Our approach leverages the common structure shared between valid examples to split the original problem into smaller sub-problems, using a divide-and-conquer approach. We use SMT solving to explore and prune the search space. Furthermore, we proposed a method to synthesise capturing groups and integer conditions over the respective captures. The capture conditions further restrict the accepted values: we validate not only the input's *format*, but also its *values*. We also described an interaction model that removes ambiguity underlying the input examples. This procedure selects the correct regex among several candidates, thus increasing confidence in the returned solution.

We implemented our approach in a tool, FOREST, and tested it in real-world regex validation benchmarks. Our experimental evaluation shows that our multi-tree representation synthesises over three times more regexes than previous representations in the same amount of time and, together with the user interaction model, FOREST solves 74% of the benchmarks with the correct user intent. When FOREST returns a solution, it is the intended regular expression in 98% of the instances. We saw that pruning the search space by removing equivalent regular expressions from consideration cuts our synthesis time in half and that our divide-and-conquer approach allows us to solve programs in two-thirds of the time, on average. We verified that FOREST maintains a very high accuracy (96%) with as few as 10

examples of each kind. Finally, We compared our approach with REGEL, a state-of-the-art synthesizer, and observed that FOREST outperforms REGEL in the domain of form validations.

In the future, FOREST can be improved in several ways. First, FOREST could ask for only two sets of examples, valid and invalid. In practice, this would make the distinction between *invalid* and *conditional invalid* examples invisible to the user. Then, either as a preprocessing step, or done dynamically during synthesis, FOREST would automatically classify the invalid examples, differentiating between examples that are invalidated by the regular expression and by the capture conditions.

Another way to make FOREST more user-friendly is by allowing some noise in the specification, both by allowing some wrongly classified examples at the beginning of synthesis, and by permitting the user to provide some wrong answers during disambiguation. Previous work has proposed to synthesise programs from noisy examples [3, 50, 53], where instead of finding a program that satisfies all examples, the goal is to *maximise* the number of examples satisfied.

Regarding the regular expressions' DSL, it could be extended in many ways. The DSL could include a broader set of character classes, such as `[,./]` to represent separators, for instance. Moreover, including UTF-8 characters in the character classes would allow for more flexibility in the synthesised expression, and more adaptability to non-English applications. Provided UTF-8 character classes are supported by the regex theory, this improvement should not bring any extra complexity to the synthesis algorithm. The synthesis of capture conditions can also include more operators, besides the inequality operators, $\leq$ and $\geq$. It would be interesting to explore the synthesis of more complex capture conditions, such as conditions depending on more than one capture. This would allow more restrictive validations; for example, referring one last time to the motivating example in Section 1.1, we could limit the possible values for the day in the date to reflect the month to which they refer, and even leap years.

More interaction models can be tried out with FOREST, especially during the synthesis of capture conditions. One could either adapt Ramos et al.'s interaction models to this domain, like we have done for regular expressions, or find some other way to maximise the number of conditions eliminated from the search space with each interaction. It might be advantageous to explore different sketching techniques for FOREST. The two techniques we tried were flawed, but they can be improved: Graph-based sketch completion can be improved by finding a way to prune the space of possible completions during the search. SMT-based completion can be improved by experimenting with variations in the encoding: either using a regex theory that allows variables inside the regular expression, like was our original intention, or simply build a less memory-consuming SMT formula. Aside from our sketching models, a completely new approach could be explored. As we saw in Section 6.1, we could have a multi-modal approach, similar to that of REGEL [8]. Particularly, REGEL's sketch generation method could be integrated into FOREST's synthesis procedure.

Finally, FOREST could be integrated in OutSystems Service Studio. OutSystems already generates forms from tables containing possible values for each input field. Right now, the users need to add hand-written validations that they would like checked before the form is submitted. FOREST can be added to this process to synthesise validations for the input fields. The values from the tables would play the role of valid examples, so FOREST would just require the user to provide additional invalid examples.

Bibliography

[1] D. Ahmed, A. Peruffo, and A. Abate. Automated and sound synthesis of lyapunov functions with SMT solvers. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2020.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[3] S. Almagor and O. Kupferman. Good-enough synthesis. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 541–563. Springer, 2020.

[4] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013.

[5] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR (Poster)*. OpenReview.net, 2017.

[6] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *FMCAD*, pages 55–59. IEEE, 2017.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press. ISBN 978-1-58603-929-5.

[8] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig. Multi-modal synthesis of regular expressions. In *PLDI*, pages 487–502. ACM, 2020.

[9] Y. Chen, R. Martins, and Y. Feng. Maximal multi-layer specification synthesis. In *ESEC/SIGSOFT FSE*, pages 602–612. ACM, 2019.

[10] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng. Program synthesis using deduction-guided reinforcement learning. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[11] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[12] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In *ICSE*, pages 345–356. ACM, 2016.

[14] K. Ellis and S. Gulwani. Learning to learn programs from examples: Going beyond program structure. In *IJCAI*, pages 1638–1645. ijcai.org, 2017.

[15] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, pages 422–436. ACM, 2017.

[16] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In *POPL*, pages 599–612. ACM, 2017.

[17] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435. ACM, 2018.

[18] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[19] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *PLDI*, pages 441–452. ACM, 2012.

[20] C. C. Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240. William Kaufmann, 1969.

[21] S. Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24. ACM, 2010.

[22] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[23] S. Gulwani and M. Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD Conference*, pages 803–814. ACM, 2014.

[24] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[25] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2): 1–119, 2017.

[26] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38. ACM, 2013.

[27] P. Huang, C. Wang, R. Singh, W. Yih, and X. He. Natural language to structured query generation via meta-learning. In *NAACL-HLT (2)*, pages 732–738. Association for Computational Linguistics, 2018.

[28] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.

[29] D. Jurafsky and J. H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.

[30] S. Kolb, S. Teso, A. Passerini, and L. D. Raedt. Learning SMT(LRA) constraints using SMT solvers. In *IJCAI*, pages 2333–2340. ijcai.org, 2018.

[31] N. Kushman and R. Barzilay. Using semantic unification to generate regular expressions from natural language. In *HLT-NAACL*, pages 826–836. The Association for Computational Linguistics, 2013.

[32] M. Lee, S. So, and H. Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *GPCE*, pages 70–80. ACM, 2016.

[33] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *Proc. VLDB Endow.*, 8(13):2158–2169, 2015.

[34] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. In *EMNLP*, pages 1918–1923. The Association for Computational Linguistics, 2016.

[35] Z. Manna and R. Waldinger. A deductive approach to program synthesis. In *IJCAI*, pages 542–551. William Kaufmann, 1979.

[36] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[37] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[38] M. H. Manshadi, D. Gildea, and J. F. Allen. Integrating programming by example and natural language programming. In *AAAI*. AAAI Press, 2013.

[39] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *Proc. VLDB Endow.*, 12(12):1914–1917, 2019.

[40] 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 *UIST*, pages 291–301. ACM, 2015.

[41] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML (1)*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 187–195. JMLR.org, 2013.

[42] K. Morton, W. T. Hallahan, E. Shum, R. Piskac, and M. Santolucito. Grammar filtering for syntax-guided synthesis. In *AAAI*, pages 1611–1618. AAAI Press, 2020.

[43] J. L. Newcomb and R. Bodík. Using human-in-the-loop synthesis to author functional reactive programs. *CoRR*, abs/1909.11206, 2019.

[44] P. Orvalho. Squares: A sql synthesizer using query reverse engineering. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 11 2019.

[45] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In *CP*, volume 11802 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2019.

[46] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Squares. `https://squares-sql.github.io`, 2019. Accessed on 27th May, 2020.

[47] 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.

[48] OutSystems. Outsystems. `https://www.outsystems.com`, 2020. Accessed on 24th October, 2020.

[49] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. In *ICLR (Poster)*. OpenReview.net, 2017.

[50] H. Peleg and N. Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *ECOOP*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[51] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *OOPSLA*, pages 107–126. ACM, 2015.

[52] Python Software Foundation. Python3's regular expression module `re`. `https://docs.python.org/3/library/re.html`, 2001-2020. Accessed on 11th October, 2020.

[53] D. Ramos. Program synthesis from noisy tabular data. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 10 2019.

[54] D. Ramos, J. Pereira, I. Lynce, V. Manquinho, and R. Martins. Unchartit: An interactive framework for program recovery from charts. In *ASE*. IEEE, 2020.

[55] D. Ramos, J. Pereira, I. Lynce, V. Manquinho, and R. Martins. Unchartit. `http://sat.inesc-id.pt/unchartit/home/`, 2020. Accessed on 24th October, 2020.

[56] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, pages 792–800. AAAI Press, 2015.

[57] Regular Expression Library. `www.regexlib.com`, 2001-2020. Accessed on 27th May, 2020.

[58] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[59] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*. Springer, 1997. ISBN 978-3-642-63863-3. doi: 10.1007/978-3-642-59136-5.

[60] D. E. Shaw, W. R. Swartout, and C. C. Green. Inferring LISP programs from examples. In *IJCAI*, pages 260–267, 1975.

[61] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2015.

[62] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.

[63] A. Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

[64] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.

[65] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[66] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.

[67] P. D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1): 161–175, 1977.

[68] M. Terra-Neves. Distributed solver for maximum satisfiability. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 10 2014.

[69] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *IJCAI*, pages 241–252. William Kaufmann, 1969.

[70] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*, pages 452–466. ACM, 2017.

[71] C. Wang, A. Cheung, and R. Bodík. Interactive query synthesis from input-output examples. In *SIGMOD Conference*, pages 1631–1634. ACM, 2017.

[72] X. Wang, S. Gulwani, and R. Singh. FIDEX: filtering spreadsheet data using examples. In *OOPSLA*, pages 195–213. ACM, 2016.

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

[74] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J. Lou, T. Liu, and D. Zhang. Semregex: A semantics-based approach for generating regular expressions from natural language specifications. In *EMNLP*, pages 1608–1618. Association for Computational Linguistics, 2018.