University of Michigan, Ann Arbor

Accelerate Regular Expresssion Synthesis via Subexpression Queries

Report of EECS499 Advanced Directed Study

Author
Zhiyang Chen

Advisor
Prof. Xinyu Wang

December 9, 2020

Contents

Chapter 1

Overview

In computer science, program synthesis is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification. After the development of first automated theorem provers, there was a lot of pioneering work on deductive synthesis approaches. The main idea behind these approaches was to use the theorem provers to first construct a proof of the user-provided specification, and then use the proof to extract the corresponding logical program. But the deductive synthesis approaches assumed a complete formal specification of the desired user intent was provided, which in many cases proved to be as complicated as writing the program itself. This lead to new inductive synthesis approaches that were based on inductive specifications including input-output examples[1], natural languages[2], and partial programs[3].

However, examples are inherently an under-specification, i.e. there might be multiple possible programs in a rich hypothesis space that are consistent with a given set of examples. Given enough examples, one can refine the specification such that only desired programs are consistent.

For example, during one regular expression synthesis task, the experimenter gave the following examples:

```
+91789  , Positive Example
91789   , Positive Example
91+     , Negative Example
9+1     , Negative Example
abc&ˆ   , Negative Example
```

Figure 1.1: The input-output examples

Even though it is clear for experts that the user's intent is to find a regular expression that accepts strings starting with an optional $+$ sign, followed by a sequence of digits. There are some simpler regular expression that can also match these positive and negative examples as Figure 1.2

The regular expressions in the Figure 1.2 perfectly matches all input-output examples, However, since there is no way the synthesizer can read users' intent apart from input-output examples, the synthesizer has to consider every possible regular expression. As

```
endwith(<9>)
contain(<8>)
contain(<7>)
repeatatleast(or(<num>,<+>),5)
repeatatleast(or(<num1-9>,<+>)),5)
```

Figure 1.2: The input-output examples

a result, the synthesizer prunes a subset of program space only if all regular expressions in that subspace cannot satisfy some of input-output examples. As a result, it spends a lot of time on all branches, therefore cannot expand to some specific directions. For another, when synthesis results return, users have to think about more counter-examples to dispute those correct but not desired regular expressions, which is a heavy workload for users.

Chapter 2

Background

2.1 Domain Specific Language(DSL)

In this project, I use the same regex domain specific languages as a previous regular expression synthesis paper[4]. The domain specific languages they defined are briefly included below:

$$
\begin{aligned}
e := {} & \texttt{startwith}(e) \mid \texttt{endwith}(e) \mid \texttt{contain}(e) \mid \texttt{concat}(e_1, e_2) \mid \texttt{not}(e) \mid \texttt{or}(e_1, e_2) \mid \texttt{and}(e_1, e_2) \\
& \mid \texttt{optional}(e) \mid \texttt{star}(e) \mid \texttt{repeat}(e, k) \mid \texttt{repeatatleast}(e, k) \mid \texttt{repeatrange}(e, k_1, k_2) \\
& \mid \texttt{<num>} \mid \texttt{<let>} \mid \texttt{<low>} \mid \texttt{<cap>} \mid \texttt{<any>} \mid \cdots \mid \texttt{<a>} \mid \texttt{<b>} \mid \cdots
\end{aligned}
$$

Figure 2.1: The DSL for regular expressions.

Many common regular expressions can be converted into this predefined domain-specific language, shown in Figure 2.1. This DSL includes *character classes* as basic building blocks. `<num>` is a character class that matches any digits from `0` to `9`. `<let>` is another character class that matches any English letters. `<low>`, `<cap>`, `<any>` match lower-case letters, upper-case letters, and any characters, respectively. In addition to these general character classes, this DSL also includes specific character classes that match only one character, e.g., `<d>` only matches letter `d`.

The DSL also includes *operator* to combine *character classes*. For example, `contain(<let>)` recognizes any strings that end with an English letter, and `star(<num>)` matches a sequence of digits of arbitrary length. This DSL provides high-level abstractions that are essentially wrappers of standard regex. This makes DSL programs more amenable to program synthesis as well as readable to users.

It is noted that in a AST tree, *character classes* are the leafs while *operators* are the nodes. Therefore every sub-tree of a AST tree also represents a regular expression.

2.2 Example of Motivation

The motivation of this project starts from a stack-overflow regular expression post. In this post, the desired regular expression is supposed to match the following eligible format:

```
(xxx)xxxxxxx
(xxx) xxxxxxx
(xxx)xxx-xxxx
(xxx) xxx-xxxx
xxxxxxxxxx
xxx-xxx-xxxxx
```

Figure 2.2: The positive examples(here x stands for any number)

The ground-truth regular expression is \(?\d{3}\)?-? *\d{3}-? *-?\d{4}.

Obviously this expression is a very complicated one. If we convert it into an AST tree, the depth of that AST tree will be more than 6 and the number of operators will be more than 10. If we simply use current state-of-the-art synthesizer like [4] to synthesize this regex, it may take hours or even days, for the reason that program space will grow exponentially w.r.t. depth.

However, some observations can be made about this complicated ground-truth regex and user given input-output examples:

- The character classes of the ground-truth regex can be easily inferred from positive examples. For example, `<(>`, `<num>`, `<->`. Additionally, it shouldn't be too hard for people to identify which character class should be included in ground-truth.
- Some parts of the ground-truth regex match certain part of positive examples. For example, `\d{3}` always matches xxx, `\(?` might match "(" in example 1, 2, 3, 4.

Based on the above observations, it is possible to find some properties about the ground-truth program via observations before the synthesis begins, and the synthesizer can use these properties to accelerate and prioritize search.

Chapter 3

Problem Formulation

As mentioned in the previous chapters, it is possible to reveal some properties of the ground-truth program before synthesis starts, and if we get these properties confirmed by users, it can largely prune the search space when doing enumeration.

Based on the observations, I find the sub-expressions of a program is easy to identify. For example, in the previous example, it is easy to observe that all positive examples contain three consecutive numbers, which indicates `\d{3}` might be contained in the ground-truth program.

Definition 1 (Sub-expression). Suppose the AST tree of a program P_0 is T_0, the AST tree of another program P_1 is T_1. P_1 is said to be a sub-expression of P_0 if and only if there exists $t \in \{subtrees\ of\ T_0\}$, such that $T_1 \equiv t$.

Since the correctness of sub-expressions need to be confirmed by users, it is necessary to limit the query times. When the number of queries are fixed, this problem can be formulated as:

Definition 2 (Sub-expression Query Problem). Suppose an oracle is able to answer observational properties about the ground-truth program. Sub-expression Query Problem can be expressed as, given a set of input-output examples and an oracle, find the maximum number of sub-expressions of the ground-truth when the number of queries is fixed.

Chapter 4

Approach

4.1 Algorithm

Algorithm 1: Sub-expression synthesis and query

```
Input: A set of input-output examples, an oracle of the ground-truth program, a
       upper limit of number of queries MaxQuery
Output: A set of sub-expressions of the ground-truth program
Queue worklist = {e};
query = 0;
List SubExpressions = {};
PriorityQueue Candidates = {};
for Depth = 1,2,3,... do
    Candidates = $inplace - synthesis$(worklist, SubExpressions)
    for c in Candicates do
        if query > MaxQuery then
            return SubExpressions
        end
        if Oracle(c) then
            SubExpressions.add(c)
            if FulfillDepth(Depth) then
                Break
            end
        end
        query = query + 1
    end
end
```

In Algorithm 1, $inplace-synthesis$ refers to enumeration using sub-expressions of the last layer, and rank using a heuristic function. For example, using the example of Figure 2.2, after the synthesizer knows in the ground-truth program, there are only three character classes, `<(>`, `<)>`, `<num>`. Then it can use these three character classes as basis to enumerate regular expressions of depth = 2, and rank the likelihood of these regular expressions using a heuristic function. An expected outcome is that regexes like `repeat(<num>,3)`, `optional(<(>)` are ranked very high.

4.2 Heuristic Function to find sub-expressions

In this project, a heuristic function is used to evaluate the likelihood of a program to be a sub-expression of the ground-truth program.

The heuristic function consists of the following components:

1. **Coverage**. Coverage refers to the extent that the positive examples are (consecutively) matched by a program. For example, suppose a positive example is "342-123-4566", and the program is `repeat(<num>,3)`. Then "342", "123", "456" match `repeat(<num>,3)`. So the coverage score of `repeat(<num>,3)` on the positive example "342-123-4566" is $\frac{9}{12} = 0.75$. The higher the coverage score is, the more likely this program is a sub-expression of the ground-truth program.

2. **Occurrence**. Occurrence refers to the number of matches in a positive example. For example, the occurrence of `repeat(<num>,3)` on the positive example "342-123-4566" is 3, since three matches are found("342", "123", "456"). The occurrence of `repeat(<num>,4)` on the positive example "342-123-4566" is 1, since only one match is found("4566"). As a result, `repeat(<num>,3)` is more likely to be a sub-expression of the ground-truth program compared to `repeat(<num>,4)`. Additionally, extra score is given to programs which have the same occurrences across all positive examples. This extra score turns out to be very effective for special character classes such as `<->`

3. **Penalty for uncertainty**. As a supplement of previous two scores, it is necessary penalty to "uncertain programs". "Uncertain programs" refers to the program that contains "uncertain character" or "uncertain operators", such as `<any>`, `<optional>`, `<not>` etc. These programs can easily match every character in all positive examples therefore get a very high score of coverage and occurrences. Therefore, a penalty is given to these "uncertain programs", decided by how many "uncertain character" and "uncertain operators" are included in that program.

4.3 Depth by depth synthesis

When using heuristic functions to compare programs in `PriorityQueue Candidates`, it is observed that the query results are not satisfying when programs of all depths are evaluated together. The reason is that complicated programs can be more flexible to match parts of positive examples. For example, the score of `or<->,<&>` is always higher than `<->` or `<>`. Thus it is not fair to compare them together. Additionally, the programs with larger depths are built by the programs with smaller depths. The programs with larger depths should only be evaluated when some of its components are confirmed to be a sub-expression of the ground-truth program.

Therefore, the idea of depth by depth synthesis is introduced in $inplace - synthesis$. The high level idea is to first find sub-expressions of depth 1, then use these sub-expressions to synthesize sub-expressions of depth 2, etc, until all query times are used up or the ground-truth program have been revealed in this query process.

Chapter 5

Evaluation

5.1 Benchmarks used

To conduct the evaluation, I used the same StackOverflow data set as REGEL[4], which was obtained by searching StackOverflow using relevant keywords such as "regex", "regular expression", "text validation" etc.

This data set contains 122 regex-related tasks and their corresponding DSL ground-truth obtained by directly converting the answer on StackOverflow to DSL. The original data set also contains both the an English description as well as positive and negative examples. But for the purpose of this project, only positive and negative examples, and the ground-truth program are used.

5.2 Evaluation Methods & Results.

To evaluate the effectiveness of Algorithm 1, I ran Algorithm 1 on REGEL benchmarks with `MaxQuery` $= 20$, `MaxQuery` $= 10$ and `MaxQuery` $= 0$, respectively. The timeout is set to be 5 minutes. I compare the number of benchmarks solved by three executions, and the time spent to find the ground-truth programs.

There are three outcomes of an execution.

- match: the algorithm successfully find the ground-truth program or its equivalence.
- not match: the algorithm find programs which satisfy all input-output examples, but none of them is equivalent to ground-truth.
- timeout: the algorithm fails to find a program which satisfies all input-output examples.

First, the outcomes of benchmarks are counted, and the results are shown in Figure 5.1, 5.2, and 5.3 respectively.

Comparing Figure 5.1 with Figure 5.2, the effect of more complicated sub-expressions exposed is evident. With 20 queries, 47.7% benchmarks are solved(mark as 'match' in Figure 5.1), whle with 10 queries, only 41.1% benchmarks are solved. The result meets my expectation. When more complicated sub-expressions are exposed, the synthesizer, using sub-expressions of larger depths, should be able to explore a program space with larger depths, therefore it will approach to the ground-truth program faster.

Another interesting observation is that, compared to 10 queries, there are more 'timeout' when 20 queries is used. It can be explained by the definitions of 'not match' and 'timeout' in the previous section. With more complicated sub-expressions exposed, these sub-expressions will be integrated into the next iteration of synthesis, which restricts the program space a lot so the synthesizer will miss some correct but not desired regexes. In other words, many programs that satisfy all input-output examples but is not equivalent to ground-truth programs are pruned as more sub-expressions are exposed.

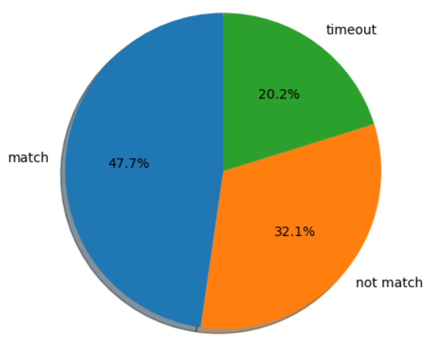

Figure 5.1: Execution with 20 queries

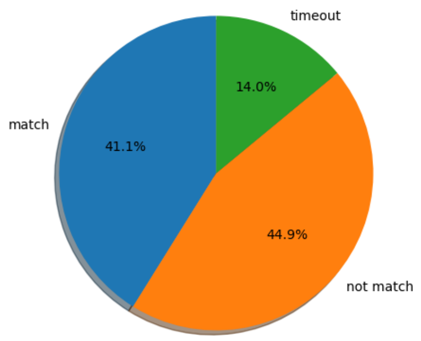

Figure 5.2: Execution with 10 queries

Comparing Figure 5.3 with Figure 5.1 and Figure 5.2, we can conclude the performance of the synthesizer becomes much better when ten or twenty queries are asked. Without any queries, the ground-truth programs are only found in 25.7% benchmarks. With ten queries, the ground-truth can be found in 41.1% benchmarks. With twenty queries asked, the ground-truth programs can be found in 47.7% benchmarks. It clearly supports the conclusion that queries about sub-expressions largely accelerate the synthesizing process.

For the benchmarks that all three execution find the ground-truth, I compare their time spent to find the ground-truth. The Figure 5.4 plots time versus the number of benchmarks solved. In Figure 5.4, at any time, the synthesizer with 20 queries solves more benchmarks than the synthesizer with 10 queries, than the synthesizer without any queries. The Figure 5.4 clearly supports the expectation that sub-expression queries largely accelerate regular expression synthesis process.

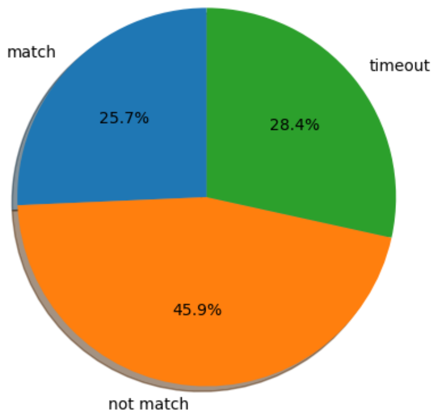

Figure 5.3: Execution without queries

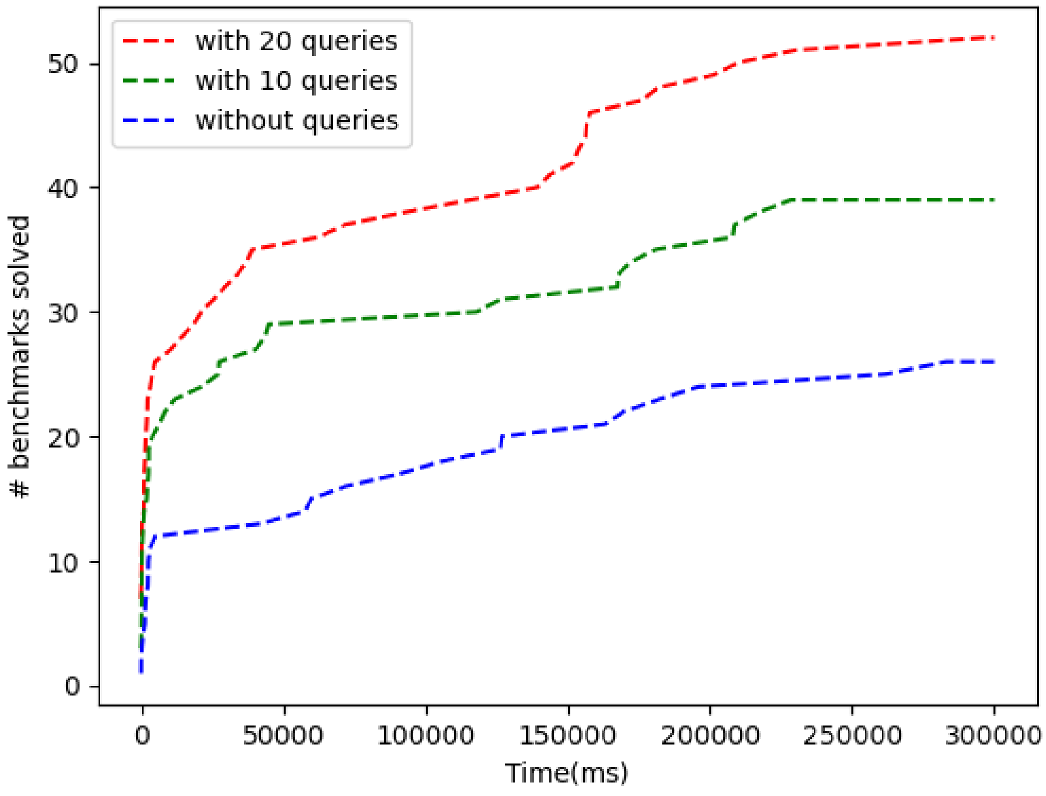

Figure 5.4: Time spent to find ground-truth program

Details about experiment results can be found via this Google Sheets.

Chapter 6

Conclusion & Future Thoughts

Overall, in this project, I proposed an algorithm to synthesize the most probable sub-expressions of the ground-truth regular expressions based on input-output examples. I evaluated and testified the effectiveness of sub-expression queries to accelerate regular expression synthesis.

In the future, we may need a more concrete interactive method for queries about sub-expressions. As sub-expressions becomes more and more complicated, it may be hard for end-users to tell whether a program is a sub-expression of another program. Other visualization methods are needed. One possible way is to highlight sub-expressions in the original input-output examples, or provide additional examples of sub-expressions to tell users the meaning of these sub-expressions.

Another problem is that so far our query is only a True/False query. If users answer 'yes', then a new sub-expression is confirmed and a lot of information is gained. But if users answer 'no', little information is gained. One possible way to use negated queries is to choose next query based on previous query results. In other words, in the next round of selecting regular expression queries, apart from the heuristic function, previous query results should also be considered.