SCALING PROGRAM SYNTHESIS: COMBINING PROGRAMS LEARNED ON SUBSETS OF EXAMPLES

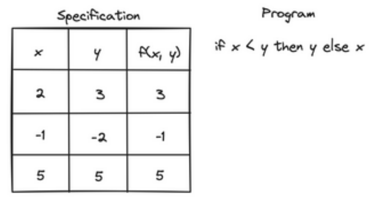
TUDelft

Author: Tudor Andrei

Supervisor: Reuben Gardos Reid
Responsible Professor: Sebastijan Dumančić

1 INTRODUCTION

- Program synthesis aims to find programs that respect some user-provided specifications. The problem of synthesizing programs is undecidable [1], with the amount of possible programs to search increasing exponentially with the amount of lines.
- Advancements in computing power, machine learning and novel algorithms have made program synthesis viable in different areas of CS such as code completion [2] and super-optimization [3].
- We examine problems where the solution needs to satisfy the examples a user provides.



Specification

x	y	f(x, y)
2	3	3
-1	-2	-1
5	5	5

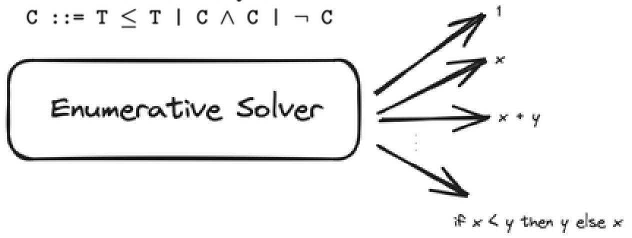
Program

if x < y then y else x

2 BACKGROUND

- In modern program synthesis, the user provides a grammar from where a synthesiser produces programs.
- One common method of solving PBE is an *Enumerative Solver*. In its simplest form it incrementally produces programs from the given grammar untill all examples are satisfied. Most of these programs are really bad choices and should be pruned.
- The enumerative solver is guaranteed to find the solution with the smallest depth, which is usually preferred.

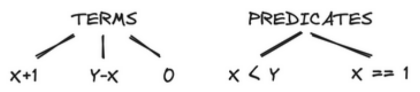
```
S ::= T | if (C) then T else T
T ::= 0 | 1 | x | y | T + T
C ::= T ≤ T | C ∧ C | ¬ C
```



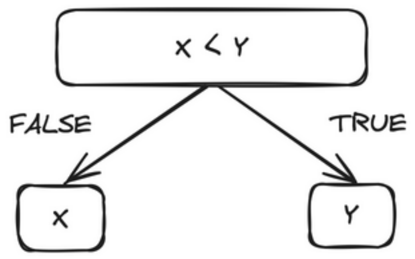
RESEARCH QUESTION

(RQ) How do we combine individual programs learned on each example to form a single program that works on many (all) examples?

- **Terms** are solutions to individual examples
- **Predicates** are the expressions in our grammar that evaluate to a boolean value.



1. There is a requirement that the **"if construct" is present in the grammar**.
2. We produce terms for **each** input-output pair. They need to collectively satisfy all examples.
3. We build a **decision tree** using **predicates as internal nodes** and **terms as leaves** such that if an input-output pair traverses the tree from the root downwards, it will end in a term that correctly solves it.
4. We convert the decision tree to a valid program using **if statements.**



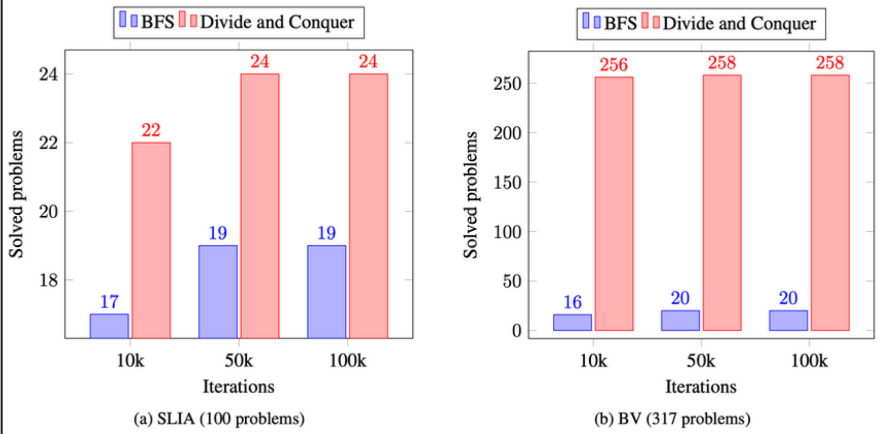


Figure 1: Performance on two datasets of problems

3 METHODOLOGY

- The presented algorithm will be implemented into Herb.jl, a program synthesis package in Julia maintained by the PONY Lab at TUDelft. In Herb, benchmarks from the SyGuS competition can be used to experiment with the Divide and Conquer method.
- The divide and conquer algorithm will be compared with the Enumerative Solver as the baseline. To evaluate one of the methods on a problem we cannot simply use the full specification. Instead we propose an evaluation scheme similar to the field of machine learning. Assuming a problem has n examples, we give the solvers $\lfloor 0.9 \cdot n \rfloor$ examples (randomly selected) to produce a program. To evaluate their solution we run it on the full specification with n examples.

4 DISCUSSION

Looking at the big difference over the two datasets we found that a revealing factor may lie in the number of examples each problem has. SLIA mostly has problems with less than 8 examples, while for the BV dataset the number of examples is at least 10. This big discrepancy might explain the difference in performance: **The more examples a problem has the harder it is for BFS to find a program that solves them all. The divide and conquer method can generate the necessary terms in much fewer iterations than BFS can find the optimal program.**

5 CONCLUSION

We found that the divide and conquer method of combining per example solutions, does improve enumerative search, but it also highly depends on it. If the enumerative search can't solve any of the examples, then neither the Divide and conquer method can. This is a more limiting factor then actually combining the programs into one, which turns out to be easier. However, our implementation has the flexibility of being able to use any method to solve the individual examples (stochastic search, machine learning), which may vastly improve this procedure.

REFERENCES

[1] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," Foundations and Trends in Programming Languages, vol. 4, no. 1-2, pp. 1–119, 2017.

[2] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in PLDI'12, June 11-16, 2012, Beijing, China, 6 2012.

[3] H. Massalin, "Superoptimizer - a look at the smallest program," in Proceedings of the Second International Conference on Architectural Support for Programming Lan-
guages and Operating Systems (ASPLOS II), (Palo Alto,California, USA), pp. 122–126, 10 1987.

T.Andrei@student.tudelft.nl