Synthesizing Optimized Programs

Abstract *Program verification* has recently become a popular means to guarantee the correctness and security of critical systems. As the complexity of those systems increased, engineers started looking into *Program Synthesis* to automate the process of implementing programs that are correct by construction. We propose extending one such program synthesis framework, namely *Syntax-Guided Synthesis (SyGuS)*, to include a new functionality of generating optimized programs according to user-provided weights. The outcomes of this work enables engineers to have access to *optimal*, *correct*, and *secure* programs that satisfy their different constraints, i.e., hardware constraints, performance constraints, etc.

1 Introduction

Our dependence on technology has increased exponentially in recent decades. We use technology in every aspect of our life, from daily mundane tasks like checking the news to life critical and expensive tasks like flying airplanes. The complexity of hardware and software also increased as a result. As we depend more on technology, there is an increasing need to ensure that those ever more complex technologies work as intended, especially in critical applications. To address this concern, engineers utilize *Program Verification* tools to formally (offering mathematical proofs) check if a program satisfies some constraints specified as logical formulas. Those constraints are encodings of the relationships between the inputs and outputs of a program. Such tools have uncovered many bugs and raised the standard of reliability to a new level that is now required in certain industries. Program Verification techniques in some fields have advanced to the point where instead of verifying that a program satisfies some correctness specifications, they can synthesize a program which satisfies those specifications by construction.

Program Synthesis has the potential to impact the quality of software even more than program verification. Program Synthesis, however, is a much harder problem to solve with a high potential to get stuck on a wrong part of the (possibly infinite) search space. To guide the solvers and limit the search space, they are often provided with a set of syntactic operators (i.e., a context-free grammar) it can compose together to generate programs in the search space (set of allowed programs). The original set of semantic constraints combined with the new set of syntactic constraints constitute a Syntax-Guided Synthesis (SyGuS) problem [1]. Figure 1 explains the basic architecture of a SyGuS solver.

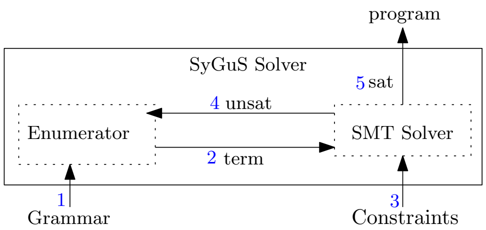

Figure 1: The diagram shows the SyGuS solver at work. Using the grammar (1), the enumerator produces a term (2) and sends it to the SMT solver. The solver checks the term against the constraints (3) and if satisfied, outputs the program (5). Otherwise, asks the enumerator for the next term (4).

2 Motivating example: matrix multiplication

Assume that we want to write a program that multiplies two matrices efficiently. Not only do we care about its functional correctness, but also about its running time. A 2×2 instance of the problem can be described as: given two matrices $A = (a_{ij})$ and $B = (b_{ij})$, find a program that efficiently computes the product, which is specified as

$$AB = \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{pmatrix} \tag{1}$$

Using a SyGuS solver to synthesize a solution, we can use the following simple grammar:

$$S ::= \begin{pmatrix} I & I \\ I & I \end{pmatrix}, \quad I ::= V \mid I + I \mid I - I \mid I * I, \quad V ::= a_{ij} \mid b_{ij},$$

where S is the start symbol. Current SyGuS solvers return a naive implementation that performs the exact operation in the formulation using 8 multiplications and 4 additions. Multiplication, however, is known to be more computationally expensive than addition. So, we would prefer implementations that reduce the number of multiplications like Strassen's algorithm, which performs only 7 multiplications, over the naive version.

3 Proposal

In this project, our main goal is to equip SyGuS solvers with extra functionality that can generate optimal programs according to some given cost constraints. First, we need to communicate to the solver the cost of each term in the search space. We can do so by adding *weight annotations* to the grammar rules of the original formulation of the SyGuS problem. For the matrix multiplication example, the extension would look like

$$S ::= \begin{pmatrix} I & I \\ I & I \end{pmatrix}^0, \quad I ::= V \mid I +^1 I \mid I -^1 I \mid I *^{16} I, \quad V ::= a_{ij}^0 \mid b_{ij}^0.$$

We then define the *total weight* of the program to be the sum of the individual weights of each rule applied to construct the program. For example, $weight(a_{11} * b_{11} + a_{12} * b_{21}) = 4 * 0 + 1 * 1 + 2 * 16 = 33$.

Adding weight annotations to grammar rules changes the original SyGuS problem into an optimization problem: synthesize a program that satisfies the (1) semantic and (2) syntactic constraints while minimizing the total (3) weight. We explore two strategies

modify enumerators to generate terms with increasing weights: current enumerators are designed to enumerate all terms in the language specified by the grammar. They start by enumerating the simplest terms (e.g., terminal symbols) and then iteratively apply production rules on them. In other words, they enumerate terms in order of the number of applied rules. Our aim here is to modify the enumerators to generate terms in order of increasing weights. This modification will force SuGuS solvers to generate minimal weight solutions that satisfy the provided semantic constraints. This change can be thought of as an extension of the current behavior of enumerators, which can be recovered by simply assigning terminal symbols a weight of 0 and other rules a weight of 1.

improve unification to generate terms with decreasing weights: unification is a process of pattern matching the structure of the desired solution against the grammar rules. If a match succeeds, the SyGuS problem is then divided into smaller sub-problems that are easier to solve. For matrix multiplication problem, we know that the solution is a 2 × 2 matrix. So, instead of enumerating all 2 × 2 matrices

$$\begin{pmatrix} a_{11} & a_{11} \\ a_{11} & a_{11} \end{pmatrix}, \begin{pmatrix} a_{11} & a_{11} \\ a_{11} & a_{12} \end{pmatrix}, \begin{pmatrix} a_{11} & a_{11} \\ a_{11} & a_{21} \end{pmatrix}, \ldots, \tag{2}$$

we can "pattern match" the rule $\begin{pmatrix} I & I \\ I & I \end{pmatrix}$ against AB and try to solve the sub-problems

$$\begin{aligned} mul(A,B)_{11} &= a_{11} * b_{11} + a_{12} * b_{21} & mul(A,B)_{12} &= a_{11} * b_{12} + a_{12} * b_{22} \\ mul(A,B)_{21} &= a_{21} * b_{11} + a_{22} * b_{21} & mul(A,B)_{22} &= a_{21} * b_{12} + a_{22} * b_{22}. \end{aligned}$$

Solving these sub-problems using enumeration is orders–of–magnitude faster than enumerating (2). State of the art solvers utilize this technique to trim the size of large search spaces. However, it not clear to us yet how to support weights with unification. For example, following a greedy approach by choosing minimum rules for pattern matching may not lead us to a minimal solution. One approach to resolve that might be to ignore weights and find an initial solution to the problem using unification. Then, iteratively use enumeration over the initial solution to reduce the weights of the sub-terms. Another open question is what the termination condition for this iterative process would be like. It might be clear for simple problems, but it gets more difficult with more complex problems. For example, such a solution does not scale for problems like matrix multiplication. A possible compromise could be to relax the weight minimality constraint to accept a range of weights as the termination condition.

4 Expected impact

Program Synthesis has a wide variety of application from synthesizing new hardware circuits, to generating SQL queries, to generating secure protocols. SyGuS utilizes the rich SMT framework and its mature tools to synthesize programs with desired correctness and security guarantees. However, it is often the case that we care about several other, sometimes competing, "soft" constraints.

Take hardware synthesis for example, a hardware architect spends much of their time minimizing cost and power consumption of their hardware while maximizing its performance (i.e., minimizing latency). We can encode those 3 constraints as weights on the grammar rules representing individual components of the hardware. We can then use SyGuS solvers to help us compare different designs that minimize each constraint independently or several combinations of those constraints.

Another example is SQL queries where the main priority is to minimize the number of disk read and writes. Disk access is often the bottleneck in slow SQL queries that operate over large datasets. We can sometimes mitigate this issue by reordering some of the operations we apply on the tables. For example, assume we have 2 tables `students` and `employees` each with 2 columns `SSN` (primary key) and `NAME` and we are looking for every person whose name starts with an `A`. We can either combine the two tables and then filter out names that do not start with an `A` or filter the two tables first and then combine them. The latter query is preferred as it reduces the number of reads and writes to disk. We can convey this information to SyGuS solvers bu giving low weights to `UNION` operations applied after `SELECT` operations and high weights for the opposite case.

TODO: come up with a security example. Either a protocol, its implementation, or maybe talk about firewalls.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.