Synthesizing Optimized Programs

1 Abstract

2 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 first-order logic 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 given 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 for a tool to get stuck on a wrong part of the infinite search space (TODO: give a simple example). To guide the tool and limit the search space the tools are often also 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. SyGuS examples generate

3 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. The problem can be described as: given 2×2 matrices A and B, then find a program that efficiently computes the product defined as

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

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

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

where S is the start symbol. SyGus will (unsurprisingly) return a naive implementation that performs the exact operation in the formulation, mainly: 8 multiplications and 4 additions. Multiplication, however, is known to be much 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.

Similar problems occur in other synthesis applications: minimizing the number of logic gates in circuits, aggregate function applications in SQL queries, etc.

One issue with the original formulation of the SyGuS problem is that it does not tell the solver the cost of using each operation. To address this issue, the SyGuS standard was recently extended to annotate grammar rules with weights:

$$T ::= a_{ij}^0 \mid b_{ij}^0, \quad I ::= T \mid I +^1 I \mid I -^1 I \mid I *^{64} I, \quad S ::= \begin{bmatrix} I & I \\ I & I \end{bmatrix}^0$$

By adding weights, there is enough information to find the optimal implementation, the one with the least weight. However, to our knowledge, no major SyGuS solver currently supports this feature.

4 Proposal

Adding weight annotations to grammar rules changes the original SyGuS problem into an optimization problem: synthesize a program that satisfies the (1) semantic (2) syntactic constraints while minimizing the total (3) weight. The main goal of our proposal is to solve this new problem by extending current techniques to account for weights:

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. Our proposal is for the enumerators to generate terms in order of increasing weight. This modification will force the SyGuS solvers to test "lighter" terms against the semantic constraints first and ensure they return a solution with minimal weight. This change can be thought as a generalization of the current behavior of enumerators, which can be recovered by assigning terminal symbols a weight of zero and other rules a weight of 1.

Improve Unification techniques 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 divided into smaller sub-problems that are easier to solve. For example, in the matrix multiplication problem, we know that the solution is a 2×2 matrix. So, instead of enumerating all 2×2 matrices

$$\begin{bmatrix} a_{11} & a_{11} \\ a_{11} & a_{11} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{11} \\ a_{11} & a_{12} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{11} \\ a_{11} & a_{21} \end{bmatrix}, \ldots$$

we can pattern match the rule $\begin{bmatrix} I & I \\ I & I \end{bmatrix}$ 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 solving the original problem. Supporting weights with unification is not as straight forward compared to enumeration. One approach is to ignore weights and find an initial solution to the problem using unification. Then, use enumeration to improve the weights bottom up. This solution does not scale for problems like matrix multiplication. However, we can use some heuristics to recover termination for some problems (e.g., rules with 0 weight like $\begin{bmatrix} I & I \\ I & I \end{bmatrix}^0$).