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 *naive* grammar:

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

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 give weights to each grammar rule:

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

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

In this research project, our goal is two fold: first to design and implement new algorithms that generate functions satisfying some optimization constraints, and second to restrict the generation of such functions based on a hint or a guide. More formally, we want to enumerate terms in the SyGuS grammar that generate functions with minimal weight. To reach such goals we are implementing two different strategies: we begin with terms having minimum weight and check them against the unweighted solution. pro: guarantees minimality of weight con: slow

The second strategy begin with the unweighted solution and try to minimize pro: fast. improves with time con: lose minimality guarantee

Practical results expected from this project: Benefits: