Synthesis and Verification of Complex Algorithms

Yican Sun[1]

Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China
sycpku@pku.edu.cn

Abstract. Designing an algorithm is important but difficult for programmers. Thus, it is intriguing to consider automatically synthesizing algorithms. Modern algorithm synthesizers are *inductive*, searching for the program that is correct over a set of examples in a designated space. Current synthesizers only handle a small class of algorithms. Thus, we first consider approaches to design new and powerful algorithm synthesizers to handle more algorithms. Moreover, since inductive algorithm synthesizers may synthesize incorrect results, we also consider approaches to verify the synthesis result.

Keywords: Program synthesis · Program verification · Algorithms

1 Research Problem

Efficiency is an everlasting pursuit in programming. The most fundamental approach to achieving efficiency is designing a fast algorithm. However, designing and implementing such an algorithm is often difficult for programmers for the following two reasons. First, designing an algorithm requires human insights. For example, to design a dynamic programming algorithm, the programmers need to find the overlapping subproblems and the optimal substructures by themselves, which significantly increases the burden of programming. Second, an algorithmic implementation is error-prone due to its high code complexity. As a result, implementing an algorithm increases the risk of program flaws.

To make algorithm design and implementation easier, the automated synthesis of algorithms (termed as *algorithm synthesis*) is an attractive solution. Algorithm synthesis is a long-standing challenge in program synthesis and has received a lot of attention [3–5, 7, 9, 12]. The task of algorithm synthesis consists of two parts: (i) a specification illustrating the expected input-output behavior of the algorithm, e.g., a naive but slow reference program or a declarative description of the user intention, and (ii) an algorithm paradigm, e.g., divide-and-conquer (D&C), or dynamic programming (DP), specifying the type of the algorithm to be synthesized. For example (Fig. 1), suppose a user is interested in synthesizing a memoization algorithm to solve the 0-1 knapsack problem [6]. Thus, the user provides the declarative description of the knapsack problem

and invokes the algorithm synthesizer. After the synthesis, the user obtains the implementation of the memoization algorithm.

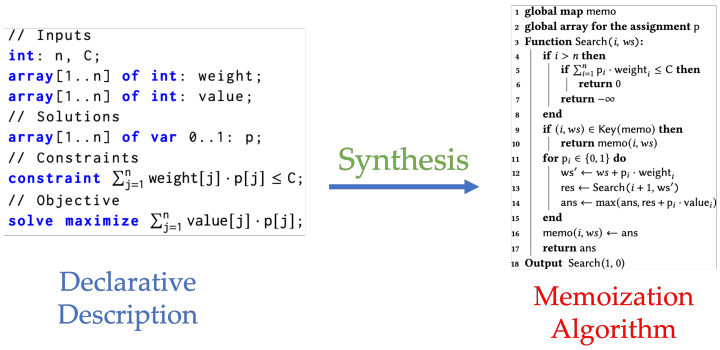

Fig. 1. An Example of Algorithm Synthesis

To effectively synthesize algorithms, modern algorithm synthesizers consider an *inductive* approach to synthesize algorithms [3–5, 9, 12]. These approaches are tailored to a specific algorithmic paradigm. They first apply the algorithmic knowledge to articulate the program space of candidate algorithms, then search all programs in this space until finding the one that is correct over a set of *input-output examples*. Using algorithmic knowledge, these approaches can effectively reduce the synthesis of the whole algorithm into the synthesis of much smaller program fragments, yielding an efficient synthesis procedure.

Though great efforts have been made in algorithm synthesis, before our work, most approaches were specific to D&C (or D&C like) algorithms [3,5], fusion [4] and a limited subclass of DP [7,9], this motivates us to propose the first question as follows.

Question 1. Can we design new and powerful algorithm synthesizers?

Moreover, inductive approaches cannot guarantee the correctness of the synthesized algorithm since we only verify the synthesis result over a set of inputs instead of all inputs. However, an algorithm is too complex to be fully verified by existing verification tools. Thus, existing inductive algorithm synthesizers still require manual inspection. To reduce the programmer's burden and make the synthesizers trustworthy, we consider the second question as follows.

Question 2. Can we automatically verify the synthesis result of algorithm synthesizers?

2 On Designing New Algorithm Synthesizers

Below, we demonstrate our approach and current state in solving Question 1.

Approach. We follow a similar discipline with previous work [3–5, 7, 9] to synthesize algorithms. To articulate the program space of algorithms, we follow a

template-based approach. The template fixes a large fraction of code, leaving some unknown program fragments to be searched. In this way, we avoid synthesizing an algorithm from scratch and alleviate the scalability issue.

The template guarantees that every program falling into it is an algorithm with the specified paradigm. The template should be carefully designed. If the template is too narrow, our approach will have limited applicability. If the template is too expressive, the synthesis problem is too difficult to solve.

Progress. We have proposed an automated approach to synthesizing efficient memoization algorithms. We implement our approach into the tool SynMem [12]. SynMem requires the user to provide a high-level declarative specification to describe the intention. The output of SynMem is a program representing the synthesized memoization algorithm, which is ensured to be in pseudo-polynomial time. An example of SynMem has been illustrated in Fig. 1. We take two novel steps to address these challenges.

First, we propose a versatile template for memoization algorithms. Our template incorporates a significantly broader class of memoization algorithms than previous works [7,9] and captures two fundamental properties of dynamic programming: the optimal substructures and the overlapping subproblems [2].

Second, by leveraging the algorithmic knowledge, we reduce synthesizing the memoization algorithms to two *independent* synthesis tasks, yielding an efficient synthesis procedure. Furthermore, the efficiency can be controlled by the range of the function synthesized by the second task. Thus, we can find the most efficient memorization algorithm by minimizing the range.

To evaluate SynMem, we collect 42 real-world benchmarks from Leetcode, the National Olympiad in Informatics in Provinces-Junior (a national-wide algorithmic programming contest in China), and previous approaches. Our approach successfully synthesizes 39/42 problems in a reasonable time, significantly outperforming the baselines [9].

3 On Verifying the Synthesis Result

This part illustrates our approach and current state in solving Question 2.

Approach. This project is divided into two steps.

1. First, we consider a specialized setting where we synthesize algorithms from a user-provided reference implementation. Furthermore, both the synthesized algorithm and the reference program are purely functional. This setting has been considered in previous works [3–5]. This setting can be treated as proving the equivalence between two functional programs, a classic and fundamental problem in program verification.
2. Second, we consider the general case (e.g., verifying the synthesis result of SynMem). Our idea is to open-box the algorithm synthesizer. We wish that the verification problem can be significantly simplified using the prior knowledge over the synthesizer.

Progress. Even the specialized setting in the first step is not easy. Proving the equivalence between two functional programs requires reasoning over algebraic data types and compositions of structural recursions, which is a difficult task in the program verification [8]. The crux of the hardness lies in lemma finding. However, there is still a lack of systematic understanding of what lemmas are needed for inductive proofs and how these lemmas can be synthesized automatically. Thus, many existing automatic proof approaches resort to *heuristic-based lemma enumeration* [1,10,11], leading to inefficiency. We present *directed lemma synthesis* to avoid enumerating useless lemmas. We propose two tactics that synthesize and apply lemmas, our tactic guarantees *progress*: it eventually produces subgoals that admit effective applications of the inductive hypothesis. The evaluation results show that, compared with the original CVC4IND, our directed lemma synthesis saves 95.47% runtime on average and help solve 38 more tasks.

References

1. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) Automated Deduction – CADE-24. pp. 392–406. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
3. Farzan, A., Nicolet, V.: Synthesis of divide and conquer parallelism for loops. ACM SIGPLAN Notices **52**(6), 540–555 (2017)
4. JI, R., ZHAO, Y., POLIKARPOVA, N., XIONG, Y., HU, Z.: Superfusion: Eliminating intermediate data structures via inductive synthesis (2024)
5. Ji, R., Zhao, Y., Xiong, Y., Wang, D., Zhang, L., Hu, Z.: Decomposition-based synthesis for applying divide-and-conquer-like algorithmic paradigms. ACM Trans. Program. Lang. Syst. (feb 2024). https://doi.org/10.1145/3648440, `https://doi.org/10.1145/3648440`, just Accepted
6. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer (2004)
7. Lin, S., Meng, N., Li, W.: Generating Efficient Solvers from Constraint Models, p. 956–967. Association for Computing Machinery, New York, NY, USA (2021), `https://doi.org/10.1145/3468264.3468566`
8. Murali, A., Peña, L., Blanchard, E., Löding, C., Madhusudan, P.: Model-guided synthesis of inductive lemmas for fol with least fixpoints. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022). https://doi.org/10.1145/3563354, `https://doi.org/10.1145/3563354`
9. Pu, Y., Bodik, R., Srivastava, S.: Synthesis of first-order dynamic programming algorithms. SIGPLAN Not. **46**(10), 83–98 (oct 2011). https://doi.org/10.1145/2076021.2048076, `https://doi.org/10.1145/2076021.2048076`
10. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 80–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
11. Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 125–148. Springer International Publishing, Cham (2021)
12. Sun, Y., Peng, X., Xiong, Y.: Synthesizing efficient memoization algorithms. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 89–115 (2023)