Scaling up Program Synthesis to Efficient Algorithms

Ruyi Ji
jiruyi910387714@pku.edu.cn
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
School of Computer Science, Peking University
Beijing, China

Abstract

The automatic synthesis of algorithms can effectively reduce the difficulty of algorithm design. However, multiple challenges exist for synthesizing algorithms. Among them, scalability of the synthesizer is the most prominent one because of the significant complexity of efficient algorithms. To address this scalability challenge, we propose several approaches from two aspects, improving the efficiency of existing program synthesizers and reducing the difficulty of algorithm synthesis by properly using algorithmic knowledge, respectively.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: Program Synthesis, Efficient Algorithms

ACM Reference Format:
Ruyi Ji. 2023. Scaling up Program Synthesis to Efficient Algorithms. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '23), October 22–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3618305.3623586

1 Motivation

Efficiency is a major pursuit in practical software development, and designing suitable algorithms is a fundamental way to achieve efficiency. Nowadays, algorithm design has become a necessary course for programmers. Various textbooks on algorithm design have been published, and various related courses have been offered. Even so, designing algorithms in practice is still difficult and even risky.

- Utilizing textbook paradigms of algorithm design is challenging for programmers because it usually requires much human intelligence. For example, the paradigm of divide-and-conquer only suggests recursively dividing the problem into sub-problems and then combining the sub-solutions obtained from sub-problems, but how to divide and combine for a concrete task is unknown and up to the programmer to discover.
- Performing algorithm-level optimizations in software design is error-prone. Such optimization may greatly increase code complexity, break modularity, and thus significantly increase the risks of program flaws.

To reduce the burden of algorithm design, the automatic synthesis of algorithms (denoted as *algorithm synthesis problems*) has attracted much research interest [1, 4, 12]. Specifically, an algorithm synthesis problem is specified by a reference program that is possibly inefficient and some available algorithmic knowledge such as a suitable algorithmic paradigm. Then, the goal of this problem is an algorithm that is not only correct (i.e., keeping the same input-output behavior as the reference program) but also as efficient as possible.

Figure 1 shows a sample algorithm synthesis problem, whose target is a parallel algorithm for calculating the second minimum in an input list. In this problem, a straightforward sort-based program is provided as the reference. It is inefficient as it runs in $O(n \log n)$ time on a list of length n. Besides, the paradigm of divide-and-conquer is also provided as it is commonly used for designing parallel algorithms. Given this problem, an algorithm synthesizer is expected to find an efficient parallel algorithm based on divide-and-conquer (the right-side program in Figure 1), which runs in $O(n/p)$ time on a list of length n and $p \leq n/\log n$ processors. In this algorithm, function *dac* deals with the sub-list in range $[l, r)$, runs in the paradigm of divide-and-conquer, and also returns the minimum of the current sub-list as an auxiliary value for merging the second minimums.

Algorithm synthesis problems can be regarded as subproblems of program synthesis [2], whose target is to automatically find a program satisfying some user-provided specification. However, existing program synthesizers can hardly solve algorithm synthesis problems because efficient algorithms are usually significantly more complex than normal programs. As shown in our sample problem (Figure 1), a correct program for the second minimum (i.e., the reference program) can be constructed within a few operators from

SPLASH Companion '23, October 22–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0384-3/23/10…$15.00
https://doi.org/10.1145/3618305.3623586

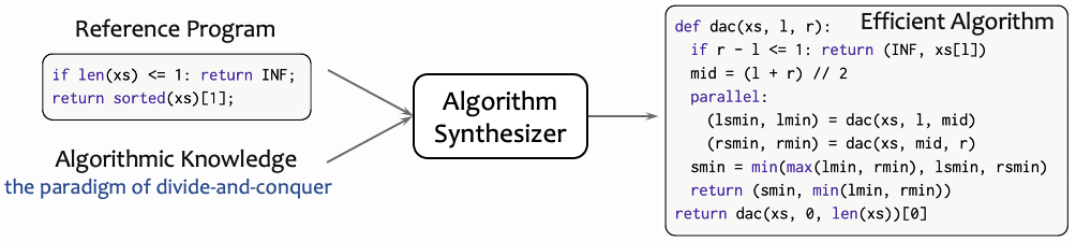

Figure 1. An algorithm synthesis problem whose target is an efficient parallel algorithm for calculating the second minimum.

library functions (i.e., *sort*). In contrast, an efficient algorithm is often constructed from a low level, resulting in a much more complex control flow and a much larger program scale. Therefore, scaling up program synthesis to support more complex programs is necessary for synthesizing algorithms.

2 Problem

Our aim is to effectively synthesize algorithms. To achieve this, we attempt to answer the following two questions.

Problem 1: Can we improve existing program synthesizers to support synthesizing programs with larger scales?

Specifically, we consider the framework of *counter-example guided inductive synthesis (CEGIS)*, a framework widely used in modern program synthesizers. As shown in Figure 2, a CEGIS solver is formed by a verifier and a *programming-by-example (PBE)* solver. Starting from an empty set of input-output examples, the PBE solver synthesizes from existing examples, and the verifier searches for a counterexample of the synthesized program. The found example will be provided to the PBE solver for subsequent synthesis, and if such an example does not exist, the current program will be returned as the final result. We aim to develop more efficient synthesizers under this framework.

Problem 2: Can we reduce algorithm synthesis problems to simpler forms by properly utilizing algorithmic knowledge?

Direct usages of algorithmic knowledge (e.g., paradigms) usually lead to complex synthesis tasks, on which efficient synthesizers (e.g., CEGIS solvers) are inapplicable. For example, a direct idea of synthesizing divide-and-conquer algorithms on lists is to fill the template in Figure 3. However, CEGIS solvers cannot be directly applied to synthesize unknown sub-programs in this template because these sub-programs are wrapped in a recursion. It is difficult to collect separate input-output examples for each unknown sub-program. We aim to develop better methods of utilizing algorithmic knowledge to make CEGIS solvers applicable.

Note that our study focuses on the efficiency of the synthesizer instead of the efficiency of the synthesized algorithms. In most cases, the latter can be easily ensured by properly designing the space of candidate programs. For example, if we want to synthesize an efficient divide-and-conquer algorithm by filling the template in Figure 3, it is enough to ensure that the program space includes only constant-time programs. At this time, the resulting algorithm must run in $O(n/p)$ time on a list of length n and $p \leq n \log n$ processors.

3 Approach

We develop several approaches to respectively answer the two questions raised in Section 2.

3.1 Improving Program Synthesis

The efficiency of a CEGIS solver depends on multiple factors, including the efficiency of the PBE solver, the quality of counter-examples, and the *generalizability* of the PBE solver (that is, the ability to find the correct program from a few examples). The first factor determines the time cost of each iteration, and the latter two determine the number of iterations required in synthesis. We propose several approaches to improve each of these factors (Figure 2).

To improve the efficiency of the PBE solver, we propose *MaxFlash* [8], which uses probabilistic models to accelerate *FlashFill* [5], one of the most successful PBE solvers. Specifically, we introduce a specialized class of models that can be effectively combined with *FlashFill*, and we also improve the search procedure of *FlashFill* to better utilize the prediction provided by the models. In this way, *MaxFlash* achieves more than 200× speed-ups compared with the original *FlashFill*.

To improve the example quality, we formalize the problem of selecting high-quality examples as *the question selection problem* and propose two effective question selectors *IntSy* [7] and *RandomSy* [6]. Concretely, we first establish a theoretical connection between the question selection problem and the problem of constructing optimal decision trees. Then, *IntSy* approximates an effective strategy for constructing decision trees by sampling, and *RandomSy* further accelerates *IntSy* by estimating the complex program semantics with simple probabilistic models. In this way, *RandomSy* can reduce the number of iterations by 4.56%-28.2% for existing CEGIS solvers with negligible extra time cost.

To improve the generalizability of the PBE solver, we establish a theory of measuring the generalizability and characterize a class of PBE solvers, named *Occam solvers*, whose generalizability is guaranteed in our theory. Then, we design an efficient Occam solver, named *PolyGen* [9], which achieves 58.2%-69.9% reductions in the number of iterations and 7.02×-15.1× speed-ups compared with previous solvers.

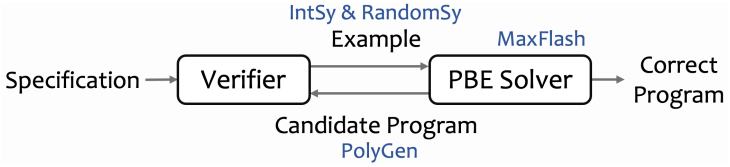

Figure 2. The CEGIS framework and the approaches we proposed for improving the components in it.

```
def dac(xs, l, r):
  if r - l <= 1: return ??
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return ??
```

Figure 3. A template of divide-and-conquer on lists.

3.2 Utilizing Algorithmic Knowledge

We investigate several common algorithmic paradigms and design specialized approaches for applying CEGIS solvers. For divide-and-conquer, we notice that to fill the template in Figure 3, it is enough to find an auxiliary program *aux* and a combinator *comb* satisfying the following formula for any lists xs_L and xs_R, where *ref* denotes the reference program, *aux* declares auxiliary values, *comb* combines the sub-results on sub-lists, and $xs_L +\!\!+ xs_R$ denotes the list concatenation.

$$\begin{aligned}&(ref\,(xs_L +\!\!+ xs_R), aux(xs_L +\!\!+ xs_R)) = \\ &\quad comb\Big((ref\,(xs_L), aux(xs_L)), (ref\,(xs_R), aux(xs_R))\Big)\end{aligned} \tag{1}$$

In the second minimum task (Figure 1), a valid *aux* returns the minimum of the list, and the *comb* calculates the first and second minimums from those of the two sub-lists.

One key observation here is that the scale of *aux* is usually much smaller than *comb* because *aux* does not need to be efficient. Therefore, the key to effectively solving Formula (1) is to apply CEGIS solvers to synthesize *comb*, which requires collecting input-output examples for *comb*. To achieve this, we propose several decomposition methods to synthesize *aux* separately before *comb*. After substituting *aux* with the synthesis result, Formula (1) will be an input-output specification for *comb*, on which CEGIS solvers are applicable.

We notice that the application of various paradigms can be reduced to a generalized form of Formula (1), and we propose *AutoLifter* [10] to solve this generalized form following the above procedure. The paradigms supported by *AutoLifter* include divide-and-conquer, single-pass, incrementalization, segment trees, and some paradigms for longest segment problems. We evaluate *AutoLifter* on 96 algorithmic tasks, and the results show that *AutoLifter* solves 82 out of 96 tasks with an average time cost of 6.53 seconds, significantly outperforming existing related approaches.

Besides *AutoLifter*, we also propose *MetHyl* [11] for synthesizing dynamic programming algorithms and *AutoElim* for eliminating intermediate data structures in inefficient programs. Similar to *AutoLifter*, these two approaches utilize domain knowledge to make CEGIS solvers applicable to synthesize the unknown parts in the resulting algorithm.

4 Evaluation Methodology

Our primary hypothesis is that we can synthesize non-trivial algorithms for algorithmic tasks. To test this hypothesis, we intend to construct a large-scale dataset of algorithm synthesis by collecting tasks from previous studies, textbooks on algorithm design such as *Introduction to Algorithms* [3], and online programming platforms such as *LeetCode* and *Codeforces*. We shall implement our approaches and evaluate their effectiveness on this dataset.

References

[1] Umut A Acar et al. 2005. *Self-adjusting computation.* Ph. D. Dissertation. Carnegie Mellon University.

[2] 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. 2013. Syntax-guided synthesis. In *FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 1–8. http://ieeexplore.ieee.org/document/6679385/

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[4] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *PLDI.* 540–555. https://doi.org/10.1145/3062341.3062355

[5] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL 2011, Austin, TX, USA, January 26-28, 2011.* 317–330. https://doi.org/10.1145/1926385.1926423

[6] Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. 2023. Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 819–847. https://doi.org/10.1145/3586055

[7] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *PLDI 2020, London, UK, June 15-20, 2020.* ACM, 1143–1158. https://doi.org/10.1145/3385412.3386025

[8] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 224:1–224:29. https://doi.org/10.1145/3428292

[9] Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485544

[10] Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2023. Divide and Conquer Divide-and-Conquer – Inductive Synthesis for D&C-Like Algorithmic Paradigms. arXiv:2202.12193 [cs.PL]

[11] Ruyi Ji, Tianran Zhu, Yingfei Xiong, and Zhenjiang Hu. 2022. Synthesizing Efficient Dynamic Programming Algorithms. *CoRR* abs/2202.12208 (2022). arXiv:2202.12208 https://arxiv.org/abs/2202.12208

[12] Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. In *OOPSLA.* ACM, 83–98. https://doi.org/10.1145/2048066.2048076

Received 2023-07-21; accepted 2023-08-10