Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms

RUYI JI, Peking University, China
YUWEI ZHAO, Peking University, China
YINGFEI XIONG∗, Peking University, China
DI WANG, Peking University, China
LU ZHANG, Peking University, China
ZHENJIANG HU, Peking University, China

Algorithmic paradigms such as divide-and-conquer (D&C) are proposed to guide developers in designing efficient algorithms, but it can still be difficult to apply algorithmic paradigms to practical tasks. To ease the usage of paradigms, many research efforts have been devoted to the automatic application of algorithmic paradigms. However, most existing approaches to this problem rely on syntax-based program transformations and thus put significant restrictions on the original program.

In this paper, we study the automatic application of D&C and several similar paradigms, denoted as D&C-like algorithmic paradigms, and aim to remove the restrictions from syntax-based transformations. To achieve this goal, we propose an efficient synthesizer, named AutoLifter, which does not depend on syntax-based transformations. Specifically, the main challenge of applying algorithmic paradigms is from the large scale of the synthesized programs, and AutoLifter addresses this challenge by applying two novel decomposition methods that do not depend on the syntax of the input program, component elimination and variable elimination, to soundly divide the whole problem into simpler subtasks, each synthesizing a sub-program of the final program and being tractable with existing synthesizers.

We evaluate AutoLifter on 96 programming tasks related to 6 different algorithmic paradigms. AutoLifter solves 82/96 tasks with an average time cost of 20.17 seconds, significantly outperforming existing approaches.

CCS Concepts: • Software and its engineering → Programming by example; • Theory of computation → Design and analysis of algorithms.

Additional Key Words and Phrases: Inductive Program Synthesis, Algorithm Synthesis, Decomposition Methods for Program Synthesis Tasks

1 INTRODUCTION

Efficiency is a major pursuit in practical software development, and designing suitable algorithms is a fundamental way to achieve efficiency. To reduce the difficulty of algorithm design, researchers have proposed many algorithmic paradigms [Mehlhorn 1984] to summarize patterns of efficient

∗Corresponding author

Authors’ addresses: Ruyi Ji, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Yuwei Zhao, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, zhaoyuwei@stu.pku.edu.cn; Yingfei Xiong, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Di Wang, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, wangdi95@pku.edu.cn; Di Wang, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, wangdi95@pku.edu.cn; Lu Zhang, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, wangdi95@pku.edu.cn; Zhenjiang Hu, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, huzj@pku.edu.cn.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/
algorithms. For example, the paradigm of divide-and-conquer (D&C) [Cole 1995] suggests recursively dividing a possibly complex problem into sub-problems and then combining the solutions for the sub-problems into the solution for the original problem.

This paper focuses on a specific class of algorithmic paradigms that share a similar idea with D&C, denoted as D&C-like algorithmic paradigms. These paradigms prescribe a recursive structure of transforming the original problem into sub-problems and aim to build up the final results step-by-step through the given recursive structure. Besides D&C, such paradigms also include (but not limited to) incrementalization [Acar et al. 2005], single-pass [Schweikardt 2018], segment trees [Lau and Ritossa 2021], and three greedy paradigms for longest segment problems [Zantema 1992].

Applying D&C-like paradigms in practice is difficult. Although these paradigms prescribe the recursive structure to build up results, how to efficiently calculate the results in each step can be significantly different among different tasks. For example, although the paradigm of D&C suggests combining the solutions for the sub-problems, how to combine these solutions in a concrete task is totally unknown and up to the developer to discover.

To reduce the burden on the user, many research efforts were devoted to automatically applying individual D&C-like paradigms, such as applying D&C [Farzan and Nicolet 2021b; Morita et al. 2007; Raychev et al. 2015], applying single-pass [Pu et al. 2011], and applying incrementalization [Acar et al. 2005]. The approaches proposed by these studies take a possibly inefficient program as input. Then, they apply their respective paradigm to the original program and aim to generate a semantically equivalent program with guaranteed efficiency.

However, the existing approaches put non-trivial restrictions on the input program, which are not easy to satisfy. Automatically applying algorithmic paradigms is difficult because optimized programs are usually complex. To cope with this challenge, most existing approaches use syntax-based program transformations. Specifically, they access the source code of the original program, transform the source code into certain forms using pre-defined program transformations, and thus simplify or even directly solve the task. However, to ensure a successful application of program transformations, these approaches put strict restrictions on the original program, leading to a significant limitation on usage. For example, approaches for D&C [Farzan and Nicolet 2017, 2021b; Morita et al. 2007; Raychev et al. 2015] require the original program to be implemented in another paradigm, namely single-pass [Schweikardt 2018]. An approach for incrementalization [Acar et al. 2005] requires the execution of the original program to be affected little by possible changes in the input, otherwise, the resulting program may not speed up, or even slow down the computation. Satisfying these requirements is typically difficult in practice. For example, in our dataset, applying single-pass already requires implementing 40.54%-58.62% of the code needed for applying D&C (Section 8.3).

In this paper, we aim to overcome the limitation of existing approaches and propose a more general approach for applying D&C-like paradigms that does not depend on syntax-based transformations. To achieve this goal, we explore another direction for addressing the scalability challenge: by decomposition. Specifically, we aim to decompose the application task into a sequence of subtasks, each corresponding to a sub-program of the original synthesis target, and solve these subtasks one by one using existing synthesizers that rely little on the source code, e.g., inductive synthesizers. Such a procedure will put little restriction on the original program if the decomposition can be accomplished without accessing the source code.

However, decomposing a synthesis task is in general difficult. In most cases, there exist mutual dependencies among different sub-programs of the synthesis target, making it impossible to derive precise specifications for independently synthesizing individual sub-programs. Our idea is to use approximate specifications in some subtasks when the precise specifications are intractable. The key point here is that, although there may be a difference between an approximation and its respective
Decomposition-Based Synthesis for Applying D&C-Like Algorithmic Paradigms

precise specification, the whole approach will still be sound if we use the original specification in the last step, and be effective when the difference is small enough.

Following the above decomposition-based idea, we propose a novel synthesizer named AutoLifter for applying D&C-like paradigms. To support applying different D&C-like paradigms, we design AutoLifter on a novel class of synthesis problems, named lifting problems, which we propose to capture the core task of applying D&C-like paradigms. We reduce the applications of various D&C-like paradigms to lifting problems, and AutoLifter can be instantiated as synthesizers for applying these paradigms.

AutoLifter decomposes lifting problems using two novel decomposition methods, namely component elimination and variable elimination, which decompose through a tuple-output structure and a function-composition structure in the specification of lifting problems, respectively. Both methods break dependency among sub-programs by producing approximate specifications in some subtasks. Consequently, during the decomposition, these methods may generate problematic sub-tasks without any valid solution, affecting the performance of AutoLifter. We investigate the effect of these approximations and provide both empirical and theoretical results showing that these approximations are precise enough to ensure the effectiveness of AutoLifter.

We conduct a thorough evaluation to verify the effectiveness of AutoLifter in applying D&C-like paradigms. Specifically, we instantiate AutoLifter as 6 inductive synthesizers, each for applying a D&C-like paradigm, including D&C, single-pass, segment trees, and the three greedy paradigms for the longest segment problem. We construct a dataset of 96 tasks for applying these paradigms, collected from existing datasets [Farzan and Nicolet 2017, 2021b; Pu et al. 2011], existing publications on formalizing algorithms [Bird 1989a; Zantema 1992], and an online contest platform for competitive programming (codeforces.com). We compare AutoLifter with existing approaches on these tasks, and our evaluation results demonstrate the effectiveness of AutoLifter.

- AutoLifter solves 82 out of 96 tasks with an average time cost of 20.01 seconds, significantly outperforming existing synthesizers that can be applied to lifting problems. Among solved tasks, the largest result includes 157 AST nodes and is found by AutoLifter in 100.0 seconds.
- AutoLifter outperforms a specialized synthesizer for applying single-pass, and when compared with an existing synthesizer for applying D&C programs, AutoLifter can offer competitive or even better performance while putting less restriction on the original program.

To sum up, this paper makes the following main contributions.

- We introduce a novel class of synthesis problems, named lifting problems, for capturing the key task of applying D&C-like paradigms (Section 3), and reduce the application tasks of various D&C-like algorithmic paradigms to lifting problems (Section 6).
- We propose an efficient approach named AutoLifter for solving lifting problems (Section 4), which decomposes lifting problems into subtasks tractable by existing inductive synthesizers with two novel decomposition methods, component elimination and variable elimination.
- We implement AutoLifter (Section 7) and evaluate it on a dataset of 96 related tasks (Section 8). The results demonstrate the advantage of AutoLifter compared with existing approaches.

2 OVERVIEW

In this section, we give an overview of our approach. Starting from an example task for calculating the second minimum of lists (Section 2.1), we discuss the synthesis task (Section 2.2), the limitation of existing approaches (Section 2.3), and the synthesis procedure of AutoLifter (Section 2.4).

For simplicity, we focus on applying the D&C paradigm in this section. The full definition of lifting problems can be found in Section 3.
2.1 Example: Divide-and-Conquer for Second Minimum

Let \( \text{sndmin} \) be a function returning the second-smallest value in an integer list. A natural implementation of \( \text{sndmin} \) (Figure 1, in Python-like syntax) first sorts the input list ascendingly and then returns the second element of the sorted list. Given a list of length \( n \), this program takes \( O(n \log n) \) time to calculate the second minimum, being inefficient.

To optimize this natural implementation, let us consider manually applying D&C, a paradigm widely used for optimization. In general, D&C decomposes a task into simpler subtasks of the same type and calculates by combining the results of subtasks. For the \( \text{sndmin} \) task, a standard procedure of D&C is to divide the input list \( xs \) into two halves \( xs_L \) and \( xs_R \), recursively calculates \( \text{sndmin} \) \( xs_L \) and \( \text{sndmin} \) \( xs_R \), and then combines them into \( \text{sndmin} \) \( xs \). In this procedure, a combinator \( \text{comb} \) satisfying the formula below is required, where \( xs_L \cdot xs_R \) represents the concatenation of two lists.

\[ \text{sndmin} (xs_L \cdot xs_R) = \text{comb} (\text{sndmin} \, xs_L, \text{sndmin} \, xs_R) \]

However, such a combinator does not exist because the second minimum of the whole list may not be the second minimum of any of the two sub-list. In the example in Figure 2, the second minimums of the sub-lists are 3 and 4, respectively, but the second minimum of the whole list is 2.

To solve this problem, a standard way is to extend the original program \( \text{sndmin} \) with a program \( \text{aux} \) (denoted as an auxiliary program) specifying necessary auxiliary values to make a valid combinator \( \text{comb} \) exist, as shown below.

\[
\text{sndmin}' (xs_L \cdot xs_R) = \text{comb} (\text{sndmin}' \, xs_L, \text{sndmin}' \, xs_R)
\]

where \( \text{sndmin}' \) \( xs \doteq (\text{sndmin} \, xs, \text{aux} \, xs) \) (1)

In this example, a valid auxiliary value is the first minimum of each sub-list. The corresponding \( (aux, comb) \) is shown in Figure 3, written in a syntax related to our synthesizer (Section 2.2). A D&C program can be obtained by filling these two programs into a template (Figure 4), where \( \text{orig} \) stands for the original program \( \text{sndmin} \) (Figure 1). In this template, function \( \text{dac} \) deals with the sub-list in range \( [l, r] \) of the input array \( xs \) and calculates the expected result (second minimum here) and the auxiliary value of this sub-list. When the sub-list contains only one element, the original program and the \( \text{aux} \) are applied directly. Otherwise, \( \text{dac} \) is recursively invoked on the two halves of the sub-list, and the results are combined by \( \text{comb} \). Note that although \( \text{aux} \) is applied only to singleton lists in this template, it is defined for all lists to guide the design of the \( \text{comb} \).
The time complexity of the resulting D&C program is $O(n)$ on a list of length $n$ when $comb$ runs in $O(1)$ time, and both $orig$ and $aux$ run in $O(1)$ time on singleton lists. This complexity can be further reduced to $O(n/p)$ on $p \leq n/\log n$ processors with proper parallelization.

As demonstrated in the above procedure, applying D&C is non-trivial. Although the template in Figure 4 is standard for D&C programs on lists, we still need to find an auxiliary program $aux$ specifying proper auxiliary values and a corresponding combinator $comb$. These programs are observably more complex than the original program in Figure 1.

### 2.2 Problem and Challenge

Motivated by the difficulty in manual optimization, we study the automatic application of D&C. Concretely, given the original program $sndmin$ (Figure 1) as the input, we aim to automatically synthesize proper $aux$ and $comb$ to fill the D&C template (Figure 4), and meanwhile ensures both the correctness and the efficiency of the resulting D&C program.

- **(Correctness)** The resulting D&C program should be semantically equivalent to the original program, that is, it should correctly calculate the second minimum of the input list. To ensure this point, we use Formula 1 as the specification for synthesizing $aux$ and $comb$. At this time, by filling the synthesized $aux$ and $comb$ to the D&C template (Figure 4), the resulting program must be correct.

- **(Efficiency)** To ensure an efficient D&C program that runs in $O(n/p)$-time in parallel, we apply the SyGuS framework [Alur et al. 2013] and constrain the space of solutions to include only $comb$ that runs in $O(1)$ time and $aux$ that runs in $O(1)$ time on singleton lists.

In this section, we use a toy solution space (Figure 5), which is simplified from the one in our implementation (Section 7), to illustrate the main idea of our approach. This solution space satisfies the constraint above, that is, every $comb$ in $\mathcal{L}^{ex}_{comb}$ runs in $O(1)$ time and every $aux$ in $\mathcal{L}^{ex}_{aux}$ runs in $O(1)$ time on singleton lists. One can verify that any possible solution $(aux, comb)$ in this toy space can lead to an efficient D&C program.

The synthesis task here is challenging because we need to synthesize two interrelated programs from a relational specification, and meanwhile the total size of these two programs can be large in real-world algorithmic problems (up to 157 AST nodes in our dataset). General program synthesis approaches that handle relational specifications such as enumerative synthesis [Alur et al. 2013] and relational synthesis [Wang et al. 2018] do not scale up to solve most of the problems in our dataset. On the other hand, most other scalable synthesis approaches [Balog et al. 2017; Feser et al. 2015; Ji et al. 2021; Miltner et al. 2022; Osera and Zdancewic 2015; Rolim et al. 2017] work only for synthesizing a single program and require obtaining input-output examples. They cannot work for synthesizing two programs from a relational specification.
2.3 An Existing Approach and Its Limitation

Parsynt [Farzan and Nicolet 2017, 2021b] is a state-of-the-art synthesizer for D&C. It solves the scalability challenge using a syntax-based program transformation system specifically designed for D&C. Specifically, Parsynt applies its transformation system to the source code of the original program to directly derive aux. After aux is derived, only comb is unknown and can be synthesized using existing synthesizers. In this procedure, Parsynt will derive the full definition of aux to help synthesize comb, though aux is invoked only on singleton lists in the D&C template (Figure 4).

The syntax-based transformation system in Parsynt puts a strict restriction on the original program, that is, the original program must be implemented as a single-pass program that enumerates each element in the input list only once. Figure 6 shows a single-pass implementation of sndmin, which is formed by a loop visiting each element in the input list xs only once. This program takes the first minimum as an auxiliary value and updates the second minimum using the property that, each time a new element is visited, the new second minimum must be the medium value among the previous first minimum, the previous second minimum, and the new element.

Although any functions that can be implemented as D&C can also be implemented as single-pass after introducing enough auxiliary values\(^1\), the single-pass restriction of Parsynt still leads to significant burdens on the user from two aspects.

- Similar to D&C, implementing single-pass programs is difficult because many functions cannot be implemented as single-pass unless auxiliary values are introduced. In the above example of sndmin in Figure 6, the user has to introduce the first minimum as an auxiliary value, which is already the auxiliary value required by D&C. In the dataset we used for evaluation, the auxiliary values required by single-pass already account for 40.54%-58.62% of the auxiliary values required by D&C (Section 8.3).
- Implementing single-pass programs is error-prone. The dataset used by Farzan and Nicolet [2021b] contains two bugs introduced when the authors manually implemented the original programs into single-pass. These bugs have been confirmed by the authors.

2.4 AutoLifter on the Second Minimum Example

To remove the requirement on single-pass original programs, we aim to solve the synthesis task without using syntax-based program transformations. Instead, we explore a decomposition-based approach to cope with the scalability challenge by answering the following question.

Is it possible to derive a specification that involves only a sub-program of the synthesis target (aux, comb), such as aux only?

Our answer is positive. We propose two decomposition methods, named component elimination and variable elimination, to derive specifications for sub-programs of the synthesis target. By applying these methods, we can first synthesize a sub-program of the synthesis target using the derived specification and then synthesize the remainder with the help of the obtained sub-program. In this way, we greatly reduce the scale of the program to be synthesized in each step. Given the difficulty of deriving a precise specification for a sub-program, we derive approximate specifications. At the end of this section we would discuss why approximate specifications do not affect the soundness of our approach.

\(^1\)By the second list-homomorphism theorem [Gibbons 1996], any function that can be implemented as D&C with a set of auxiliary values can also be implemented as single-pass with the same set of auxiliary values.
Figure 7 shows the workflow of AutoLifter, which synthesizes through an interaction between a decomposition system and two inductive synthesizers. Given a synthesis task, the decomposition methods are iteratively applied to decompose the task and in this process, we would obtain a series of leaf subtasks (i.e., subtasks that cannot be further decomposed), each with a smaller scale and a simpler form. AutoLifter solves these subtasks one by one using inductive synthesizers and collects the results for generating subsequent subtasks and constructing the final result.

**Component elimination.** In the original specification (Formula 1), the output of comb is a pair of two components, corresponding to the expected output of sndmin and the auxiliary values defined by aux, respectively. Accordingly, we can synthesize two sub-programs, denoted as comb\(_1\) and comb\(_2\), each for calculating an output component, and then constructs comb as follows.

\[
\text{comb in} \triangleq (\text{comb}_1 \text{ in}, \text{comb}_2 \text{ in})
\]

A natural idea here is to synthesize the two sub-programs individually [Osera and Zdancewic 2015]. However, this method does not work here because the two sub-programs are both dependent on aux. Specifically, we could have the specifications below for the two sub-programs, where for clarity, we use blue to denote the original program, red to denote unknown programs to be synthesized, and green to denote universally quantified variables that range over all integer lists.

\[
sndmin (x_{SL} \oplus x_{SR}) = comb_1 (sndmin' x_{SL}, sndmin' x_{SR}), \text{where } sndmin' x_{SL} \triangleq (sndmin x_{SL}, aux x_{SL})
\]

(2)

\[
aux (x_{SL} \oplus x_{SR}) = comb_2 (sndmin' x_{SL}, sndmin' x_{SR}), \text{where } sndmin' x_{SL} \triangleq (sndmin x_{SL}, aux x_{SL})
\]

(3)

Both specifications involve the same unknown program aux. If we synthesize from these specifications individually, we may get two incompatible results that use different aux.

To solve this problem, we analyze the requirement on aux put by each specification.

- In Formula 2, aux needs to provide enough information for calculating the second minimum.
- In Formula 3, aux needs to ensure that the auxiliary values provide enough information for calculating themselves.
Our observation here is that, for the first requirement, it does not matter if \( \text{aux} \) provides more information than necessary. Consequently, these two requirements can be satisfied in order. We can first find \( \text{aux} \) to satisfy the first requirement and then if necessary, add more auxiliary values to satisfy the second requirement. At this time, the first requirement will still be satisfied because more information is provided.

Following this idea, we design our first decomposition method component elimination. The decomposition procedure is shown below.

- The first subtask is the same as Formula 2. It targets finding those auxiliary values necessary for the second minimum and the corresponding combinator. Here, one possible result is to take \( \text{aux} \) as min and take \( \text{comb}_1 \) as the \( \text{csmin} \) expression in Figure 3.
- Then, the second subtask is shown below. It aims to expand the found auxiliary value to satisfy the second requirement. In this specification, the new auxiliary program \( \text{aux}' \) denotes the new auxiliary values needed to calculate auxiliary values themselves.

\[
\begin{align*}
\text{aux} (x_{SL} \oplus x_{SR}) &= \text{comb}_2 (\text{sndmin}' x_{SL}, \text{sndmin}' x_{SR}) \\
\text{where} \quad \text{sndmin}' x_S &= (\text{sndmin} x_S, \text{aux} x_S) \\
\text{aux} x_S &= (\text{min} x_S, \text{aux}' x_S)
\end{align*}
\]

In the \( \text{sndmin} \) example, no other information is needed for calculating the auxiliary value of the first minimum. One possible result here is to take \( \text{aux}' \) as an empty program (i.e., returns an empty tuple) and take \( \text{comb}_2 \) as the \( \text{aux} \) expression in Figure 3.

We shall discuss another example where new auxiliary values are needed in Section 4.
- By merging the above results of subtasks, we can obtain the intended solution in Figure 3. \( \text{AutoLifter} \) will further decompose both subtasks to achieve efficient synthesis.
- The first subtask (Formula 2) will be decomposed by another decomposition method, variable elimination, which will be introduced later.
- The second subtask (Formula 4) will be recursively decomposed by component elimination. Specifically, this subtask is similar to the original task (Formula 1) in form. Both tasks are about finding new auxiliary values to calculate the output of a known function (together with the auxiliary values themselves). Therefore, this subtask can still be decomposed similarly.

**Variable elimination.** The first task generated by component elimination is still challenging because it involves two unknown programs, \( \text{comb}_1 \) and \( \text{aux} \). Our second decomposition method variable elimination decomposes this task by deriving a subtask involving only \( \text{aux} \). In other words, this method eliminates a program variable (i.e., \( \text{comb}_1 \)) from the specification.

To derive a specification only for \( \text{aux} \), we first revisit the fundamental reason why \( \text{aux} \) is needed. Let us consider two pairs of lists, \((x_{SL}, x_{SR})\) in Figure 2 and another pair \((x'_{SL}, x'_{SR})\).

\[
\begin{align*}
x_{SL} &= [1, 3, 5], [2, 4, 6] \quad x_{SR} = x'_{SL} = [1, 3, 5], [1, 4, 6] \quad x'_{SR}
\end{align*}
\]

Although the second minimums of \( x_{SL} \) and \( x_{SR} \) (3 and 4) are the same as their counterparts of \((x'_{SL}, x'_{SR})\), the second minimum of the combined list \( x_{SL} \oplus x_{SR} \) (which is 2) differs from that of \( x'_{SL} \oplus x'_{SR} \) (which is 1). Consequently, if \( \text{aux} \) is not involved, a conflict will emerge after substituting these two list pairs into the specification of \( \text{comb}_1 \) (Formula 2). Specifically, \((x_{SL}, x_{SR})\) requires \( \text{comb}_1 \) to output 1 from input (3, 4) but \((x'_{SL}, x'_{SR})\) requires \( \text{comb}_1 \) to output 2 from the same input. Such a \( \text{comb}_1 \) does not exist because it must produce the same output from the same input.

The above analysis indicates that a necessary condition on \( \text{aux} \) is to ensure that a function exists for \( \text{comb}_1 \) to implement. When the expected outputs of \( \text{comb}_1 \) (i.e., the second minimums of the combined lists) are different, the inputs (i.e., the second minimums or the auxiliary values on the two halves) must also be different. Our method variable elimination takes this necessary condition

as the specification for synthesizing \( \text{aux} \) and thus separates the synthesis of \( \text{aux} \) and \( \text{comb}_1 \). This method decomposes the first task of component elimination (Formula 2) as follows.

- The first subtask is shown below. It targets finding auxiliary values such that two inputs of \( \text{comb}_1 \) must be different when their respective outputs differ.

\[
\text{sndmin} (x_{SL} \oplus x_{SR}) \neq \text{sndmin} (x'_{SL} \oplus x'_{SR})
\]

\[
\rightarrow (\text{sndmin}' x_{SL}, \text{sndmin}' x_{SR}) \neq (\text{sndmin}' x'_{SL}, \text{sndmin}' x'_{SR})
\]

where \( \text{sndmin}' x \equiv (\text{sndmin} x, \text{aux} x) \)

For clarity, we transform this specification into the following equivalent form to make the constraint on \( \text{aux} \) clear. One possible result here takes \( \text{aux} \) as \( \text{min} \).

\[
(sndmin x_{SL}, sndmin x_{SR}) = (sndmin x'_{SL}, sndmin x'_{SR})
\]

\[
\wedge sndmin (x_{SL} \oplus x_{SR}) \neq sndmin (x'_{SL} \oplus x'_{SR})
\]

\[
\rightarrow (aux x_{SL}, aux x_{SR}) \neq (aux x'_{SL}, aux x'_{SR})
\]

This subtask will be used for synthesis without further decomposition; in other words, it is a leaf subtask of the decomposition. In this section, we include leaf subtasks in framed boxes to distinguish them from the other tasks.

- Then, the second subtask is shown below. It aims to synthesize a corresponding \( \text{comb}_1 \) that calculates the second minimum using the auxiliary value found in the first subtask.

\[
\text{sndmin} (x_{SL} \oplus x_{SR}) = \text{comb}_1 ((\text{sndmin} x_{SL}, \text{min} x_{SL}), (\text{sndmin} x_{SR}, \text{min} x_{SR}))
\]

One possible result here takes \( \text{comb}_1 \) as the \( \text{csmin} \) expression in Figure 3.

- By merging the results of the above subtasks, we can obtain a valid solution to the original task (Formula 2).

**Synthesis from leaf tasks.** After applying the above two decomposition methods, the original synthesis task (Formula 1) is decomposed into two series of leaf tasks, one for sub-programs of \( \text{aux} \) (represented by Formula 6) and the other for sub-programs of \( \text{comb} \) (represented by Formula 7). We solve these leaf tasks following the framework of counter-example guided inductive synthesis (CEGIS) [Solar-Lezama 2013]. In CEGIS, the synthesizers focus on satisfying a set of examples (i.e., instances of the quantified variables \( x_{SL}, x_{SR}, x'_{SL}, \) and \( x'_{SR} \)) instead of the full specification, and a verifier verifies the correctness of the program synthesized from examples and provides new counter-examples when it is incorrect.

Among the leaf tasks, the task for sub-programs of \( \text{comb} \) (e.g., Formula 7) is already in the input-output form, where input-output examples are available. For example, under example \( (x_{SL}, x_{SR}) \triangleq ([1, 3, 5], [2, 4, 6]) \), Formula 7 requires sub-program \( \text{comb}_1 \) to output 2 from input \((3, 1), (4, 2)\). As a result, these tasks can be solved by existing inductive synthesizers that rely on input-output examples. We use a state-of-the-art synthesizer PolyGen [Ji et al. 2021] in our implementation.

In contrast, the task for sub-programs of \( \text{aux} \) (e.g., Formula 6) is still relational. It involves the outputs of \( \text{aux} \) on four different inputs \( (x_{SL}, x_{SR}, x'_{SL}, x'_{SR}) \), making input-output examples unavailable. Fortunately, a domain property here is that the size of \( \text{aux} \) is usually much smaller than \( \text{comb} \). Specifically, since \( \text{aux} \) will only be invoked on singleton lists in the resulting program (Figure 4), it does not need to be efficient and thus can be synthesized compactly using high-level list operators. Using this property, we solve the leaf tasks for \( \text{aux} \) using observational equivalence [Alur et al. 2013], a general enumeration-based synthesizer, and also proposes a specialized pruning.
method, named *observational covering*, to speed up the synthesis for the cases requiring multiple auxiliary values. The details on this synthesizer can be found in Section 4.3.

**Notes.** There are two points worth noting in the synthesis procedure.

- Although the full definition of *aux* is not used in the resulting D&C program, it reduces the difficulty of synthesizing *comb*. As we can see, after the full *aux* is synthesized, the subsequent subtasks for *comb* are no longer relational and can be solved easily.
- Neither *PolyGen* nor OE can be directly applied to the original problem (Formula 1) without the decomposition. For *PolyGen*, neither input-output examples of *aux* nor those of *comb* can be extracted from Formula 1; and for OE, the target *comb* is too large to be efficiently synthesized by enumeration.

**Properties of AutoLifter.** The decomposition of *AutoLifter* is *sound* in the sense that any solution constructed from valid sub-programs for the leaf subtasks must satisfy the full specification. This is because, in each decomposition, the second subtask is always obtained by putting the result of the first subtask into the original specification before the decomposition. Therefore, the original specification must be satisfied when the second subtask is solved successfully.

However, both decomposition methods in *AutoLifter* are approximate, possibly decomposing a realizable task (i.e., a task whose valid solution exists) into unrealizable subtasks. This is because, both decomposition methods use approximate specifications in their first subtask, and thus it is possible to synthesize a sub-program from the first subtask that can never form a valid solution, making the corresponding second subtask unrealizable. For example, the subtask for *aux* (Formula 6) only ensures that a function exists for *comb* to implement but does not ensure that such a program exists in the program space (*L*$_{aux}^{ex}$, Figure 5b). One can verify that (*min xs*) + (*min xs*) is also valid for this subtask, but a corresponding combinator does not exist in *L*$_{comb}^{ex}$.

There are two possible strategies for using such approximate decomposition methods in practice.

- (Greedy strategy) In each decomposition, consider only the first sub-program synthesized from the first subtask and then focus only on the corresponding second subtask.
- (Backtracking strategy) Each time an unrealizable subtask is met, backtrack to the previous decomposition step and try other valid sub-programs to the first subtask.

Both strategies are effective only when the approximation is precise enough to ensure that unrealizable subtasks are seldom generated. Otherwise, the greedy strategy will be frequently stuck into an unrealizable subtask, significantly harming the effectiveness; and the backtracking strategy will frequently roll back and switch to other search branches, significantly harming the efficiency.

Fortunately, our evaluation results suggest that our decomposition methods are precise enough: they never generate any unrealizable subtask from realizable tasks in our dataset. In the remainder of this section, we shall intuitively discuss why this happens on the *sndmin* example.

To ensure that no unrealizable subtask is generated when solving the *sndmin* task, the key is to ensure that *aux* is exactly synthesized as *min xs* from the first subtask (Formula 6), given that the subsequent two subtasks (Formulas 7 and 4) are both determined by this result. *AutoLifter* achieves this through a combined effect between the enumeration-based synthesizer (mainly OE) and the program space (*L*$_{aux}^{ex}$, Figure 5a).

Programs in *L*$_{aux}^{ex}$ can be divided into two categories. The first includes programs *derived* by the intended solution *min xs*, for example, by including more auxiliary values (e.g., (*min xs*, *max xs*)) or performing some arithmetic operations (e.g., (*min xs*) + (*min xs*)). Although many programs in this category satisfy the specification (Formula 6) as well, the principle of *Occam’s razor* [Blumer et al. 1987; Ji et al. 2021] applies here: the intended solution *min xs* is the smallest in this category.
Table 1. Counter-examples of \( \text{sum} \, \text{xs} \) and \( \text{max} \, \text{xs} \) for Formula 6.

<table>
<thead>
<tr>
<th>Program</th>
<th>((x_{L}, x_{R}))</th>
<th>((x'<em>{L}, x'</em>{R}))</th>
<th>Simplified Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{sum} , \text{xs} )</td>
<td>([0, 2], [0, 1, 2])</td>
<td>([0, 2], [1, 1, 1])</td>
<td>((2, 1) = (2, 1) \land 0 \neq 1 \rightarrow (2, 3) \neq (2, 3))</td>
</tr>
<tr>
<td>(\text{max} , \text{xs} )</td>
<td>([0, 2], [0, 2])</td>
<td>([0, 2], [1, 2])</td>
<td>((2, 2) = (2, 2) \land 0 \neq 1 \rightarrow (2, 2) \neq (2, 2))</td>
</tr>
</tbody>
</table>

Since \(\text{AutoLifter}\) synthesizes \(\text{aux}\) by enumerating programs from small to large, it prefers smaller programs and thus can successfully find \(\text{min} \, \text{xs}\) from those unnecessarily complex programs.

The second category includes the remaining programs not related to \(\text{min} \, \text{xs}\). The specification for \(\text{aux}\) (Formula 6) is strong enough to exclude all programs in this category because of a property of these functions, which we name as the \textit{compressing} property. As a side effect of ensuring an efficient D&C program, program space \(\mathcal{L}_{\text{aux}}^{\text{ex}}\) includes only programs mapping a list (whose size is unbounded) to a constant-sized tuple of integers. Such programs \textit{compress} a large input space to a much smaller output space\(^2\) and thus frequently output the same on different inputs. Consequently, an incorrect program in \(\mathcal{L}_{\text{aux}}^{\text{ex}}\) can hardly satisfy the specification (Formula 6) because this specification requires \(\text{aux}\) to generate different outputs on a series of input pairs. For example, \(\text{sum} \, \text{xs}\) and \(\text{max} \, \text{xs}\) are two candidates in \(\mathcal{L}_{\text{aux}}^{\text{ex}}\) that are not related to \(\text{min} \, \text{xs}\). Both of them are rejected by Formula 6, and the corresponding counter-examples are listed in Table 1.

Note that the specification for \(\text{aux}\) (Formula 6) may be weak without the compressing property because it only requires \(\text{aux}\) to output differently on some pairs of inputs. It accepts all programs that seldom output the same, such as the identity program \(\text{id} \, \text{xs} \triangleq \text{xs}\).

The above two factors will be revisited formally in Section 5.

- First, we prove that the probability for \(\text{AutoLifter}\) to generate an unrealizable subtask converges to 0 under a probabilistic model where the semantics of programs are modeled as independent random functions with the compressing property (Theorem 5.6).
- Second, we prove that \(\text{AutoLifter}\) can always find a minimal auxiliary program (i.e., no strict sub-program of the synthesized auxiliary program is valid), helping avoid unnecessarily complex solutions when the dependency among semantics is considered (Theorem 5.8).

### 3 LIFTING PROBLEM

Section 2 shows how \(\text{AutoLifter}\) works for applying the D&C paradigm. In this section, we show how to capture the application tasks of D&C-like paradigms uniformly as \textit{lifting problems}, a novel class of synthesis tasks considered by \(\text{AutoLifter}\).

#### 3.1 Example: Incrementalization for Second Minimum

We use the paradigm of incrementalization as an example. Suppose now a series of changes are going to be applied to a list, each time a new integer will be appended, and the task is to determine the second minimum of the new list after each change. The incrementalization paradigm suggests computing some auxiliary values such that the new result after each change can be incrementally calculated from the previous one. In other words, we need to find a program \(\text{aux}\) for specifying auxiliary values and a combinator \(\text{comb}\) for quickly updating the result, as shown below.

\[
\text{sndmin'} \ (\text{append} \ \text{xv}) = \text{comb} \ (v, \text{sndmin'} \ \text{xv}), \text{where sndmin'} \ \text{xs} \triangleq (\text{sndmin} \ \text{xs}, \text{aux} \ \text{xs}) \tag{8}
\]

\(^2\)Here we assume the integer range is fixed for simplicity. The effect of the integer range on the compressing property will be discussed in Section 9.
A valid solution to this specification is shown in Figure 8. Similar to the D&C case, the auxiliary value is still the minimum element of the list, and the combinator updates both the second minimum and the first minimum with the newly appended integer \( v \). This program takes \( O(1) \) time for each update, but it is not easy to write since a proper auxiliary value is required.

We can see that the above example task of applying incrementalization has many commonalities compared with the previous example task of applying D&C to \( \text{sndmin} \) (Section 2.1).

- In both tasks, a list is created from existing lists via an operator (\( \text{xssl} + \text{xs}r \) and \( \text{append} \text{xs} \text{v} \)) and the output of an original program (\( \text{sndmin} \)) on the created list is calculated.
- Both tasks aim at finding (1) a program \( \text{aux} \) (denoted as an auxiliary program) for specifying auxiliary values, and (2) a corresponding combinator \( \text{comb} \) for calculating the outputs on the created list from those on the existing lists.

We denote such a problem as a lifting problem. As we shall demonstrate later, the auxiliary program and the original program form a homomorphism that preserves a given operation; in other words, the auxiliary program lifts the original program to be a homomorphism.

### 3.2 Lifting Problem

**Notations.** In this paper, we regard a type as a set of values of the type and use the two terms interchangeably. To distinguish between types and values, we use uppercase letters such as \( A, B \) to denote types, and lowercase letters (or words) such as \( a, x, \) \( \text{func} \) to denote values and functions. Particularly, we use overline letters such as \( \overline{a} \) to denote values in the form of tuples.

To operate types and functions, we use \( T^n \) to denote the \( n \)-arity product \( T \times \cdots \times T \) of type \( T \), \( \text{func}_1 \circ \text{func}_2 \) to apply two functions to the same value, \( \text{func}_1 \times \text{func}_2 \) to apply two functions to the two components in a pair, and \( \text{func}^n \) to apply a function to each component in an \( n \)-tuple.

\[
\begin{align*}
(\text{func}_1 \circ \text{func}_2) \; x \equiv (\text{func}_1 \; x, \text{func}_2 \; x) \\
(\text{func}_1 \times \text{func}_2) \; (x_1, x_2) \equiv (\text{func}_1 \; x_1, \text{func}_2 \; x_2) \\
\text{func}^n \; (x_1, \ldots , x_n) \equiv (\text{func} \; x_1, \ldots , \text{func} \; x_n)
\end{align*}
\]

**Problem definition.** Given an original program \( \text{orig} \) over some data-structure type and an operator that creates an instance \( a \) from some other instances \( a_1, \ldots , a_n \) of the data structure, a lifting problem is to find an auxiliary program and a combinator such that \( \text{orig} \; a \) can be calculated from \( \text{orig} \; a_1, \ldots , \text{orig} \; a_n \). Formally, a lifting problem is defined as follows.

**Definition 3.1 (Lifting Problem).** A lifting problem is specified by the following components.

- An original program \( \text{orig} \), whose input type is denoted as \( A \).
- An operator \( \text{op} \) with input type \( C \times A^n \) and output type \( A \) for some type \( C \) and arity \( n \). It constructs an \( A \)-element from \( n \) existing \( A \)-elements and a complementary input in \( C \).
- Two domain-specific languages \( \mathcal{L}_{\text{aux}} \) and \( \mathcal{L}_{\text{comb}} \), each specified by a grammar and the corresponding interpretations (i.e., semantics), defining the spaces of candidate programs.

The task of a lifting problem is to find an auxiliary program \( \text{aux} \in \mathcal{L}_{\text{aux}} \) and a combinator \( \text{comb} \in \mathcal{L}_{\text{comb}} \) such that the formula below is satisfied for any \( c \in C \) and \( \overline{a} \in A^n \):

\[
(\text{orig} \circ \text{aux}) \; (\text{op} \; (c, \overline{a})) = \text{comb} \; (c, (\text{orig} \circ \text{aux})^n \overline{a})
\]

(9)

Following the notations in Section 2, we mark the known programs given in the synthesis task (e.g., original program \( \text{orig} \) and operator \( \text{op} \) as blue, the unknown programs to be synthesized as
Table 2. The correspondence between the lifting problem and previous synthesis tasks.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Specification</th>
<th>orig</th>
<th>A</th>
<th>op</th>
<th>n</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>D&amp;C</td>
<td>Formula 1</td>
<td>sndmin</td>
<td>List</td>
<td>( \text{op} (((), (xs_L, xs_R))) ) ( \triangleq xs_L + xs_R ) &amp; 2 &amp; Unit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>incrementalization</td>
<td>Formula 8</td>
<td>op</td>
<td>(c, (x)) = append xs c &amp; 1 &amp; Int</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

red, and those universally quantified values as green. Furthermore, we shall also omit the range of a universally quantified value (such as \( \forall a \in A^n \) and \( \forall c \in C \) here) if it is clear from the context.

**Example 3.2.** The specification of a lifting problem can be transformed into the following equivalent form to better correspond to the previous synthesis tasks (Formulas 1 and 8).

\[
\text{orig} \left( \text{op} \left( c, (a_1, \ldots, a_n) \right) \right) = \text{comb} \left( c, (\text{orig}^\prime a_1, \ldots, \text{orig}^\prime a_n) \right), \quad \text{where} \quad \text{orig}^\prime x \triangleq (\text{orig} x, aux x)
\]

Table 2 associates the concepts in a lifting problem with the two previous tasks, where Unit is a singleton type, and () is the only element in Unit that provides no information.

A lifting problem is defined as a syntax-guided synthesis (SyGuS) problem [Alur et al. 2013], where the two languages \( L_{aux} \) and \( L_{comb} \) can be used to control the complexity of the generated program. We have seen the case of D&C in Section 2.2, and the incremental program generated by solving the respective lifting problem (Formula 8) must run in \( O(1) \) time per change if \( L_{comb} \) includes only programs running in constant time.

In this paper, we assume that \( L_{aux} \) and \( L_{comb} \) are implicitly given and denote a lifting problem as \( LP(\text{orig}, \text{op}) \). Besides, we assume that \( L_{aux} \) contains a constant function \( null \) mapping anything to the unit constant (), corresponding to the case where no auxiliary value is required.

**Meaning.** A lifting problem has a clear algebraic meaning about synthesizing a homomorphism. For clarity, we draw the commutative diagram of its specification on the right, where each arrow represents a function application and the two paths from the lower-left to the upper-right result in the same function, \( id \) is the identity function, and \( B \) denotes the output type of \( \text{orig} \circ aux \). This diagram shows that \( \text{orig} \circ aux \) is a homomorphism mapping from \( A \) to \( B \), where the operator \( \text{op} \) (related to \( A \)) is preserved as the combinator \( \text{comb} \) (related to \( B \)).

In the sense of program optimization, the lifting problem is about eliminating the construction of \( A \)-elements. In its specification (Formula 9), the left-hand side explicitly constructs an \( A \)-element via \( \text{op} \) and immediately consumes it via \( \text{orig} \); in contrast, the right-hand side avoids this construction by directly calculating from existing results, via the synthesized combinator \( \text{comb} \). This intuition matches a general optimization strategy, named fusion [Pettorossi and Proietti 1996], which suggests that a program is efficient if there is no unnecessary intermediate data structure produced and consumed during the computation.

**Applying to D&C-like algorithmic paradigms.** Besides D&C and incrementalization, there are many other algorithmic paradigms sharing the idea of building up the final results step-by-step through a prescribed recursive structure. We denote these paradigms as D&C-like paradigms, and other such paradigms include single-pass [Schweikardt 2018], segment trees [Lau and Ritossa 2021], and three greedy paradigms for longest segment problems [Zantema 1992]. The application of these paradigms can also be reduced to lifting problems, as shall be discussed in Section 6.

Given a reduction from the application of a certain D&C-like paradigm to lifting problems, any synthesizer for lifting problems can be instantiated as a synthesizer for applying the respective
paradigm. In practice, to obtain an efficient algorithm for a specific task, the user needs only to select an available D&C-like algorithmic paradigm, pick up the corresponding instantiated synthesizer, and provide the original program. Then, the instantiated synthesizer will automatically generate a semantically equivalent program in the target paradigm.

We shall discuss how to select an algorithmic paradigm in Section 9.

4 APPROACH

In this section, we shall illustrate AutoLifter in detail with a more complex example related to a classic problem, maximum segment sum (mss) [Bird 1989b]. This section is organized as follows. Section 4.1 introduces the mss example, Section 4.2 discusses the decomposition methods, Section 4.3 shows how to solve the leaf subtasks via inductive synthesis, and Section 5 summarizes the theoretical properties of AutoLifter.

4.1 Example: Divide-and-Conquer for Maximum Segment Sum

Given a list of integers, we can create many contiguous subsequences, called segments. For each segment, we can add up integers within the segment to get the segment sum. The mss problem is to find, for a given list, the greatest sum we can get among all segments. A natural implementation of mss (Figure 9) enumerates all segments, calculates their segment sums, and returns the maximum. This program runs in \(O(n^3)\) time on a list of length \(n\) and thus is quite inefficient.

D&C can be applied to optimize this natural implementation. However, similar to the second minimum example, if we divide the input list into two halves, the mss of the whole list cannot be calculated from those of the two halves. In the case shown in Figure 10, the segment with the maximum sum of the left half is the prefix list \([3, -4, 1, 1]\), that of the right half is the prefix list \([1, 2, -5, 4]\), but the segment with the maximum sum of the whole list (i.e., \([1, 1, 1, 2]\)) is a concatenation of a tail-segment of the left half and a prefix of the right half.

To resolve the issue exposed by Figure 10, we can take the maximum prefix sum (mps) and the maximum tail-segment sum (mts) as auxiliary values so that the mss of the whole list in Figure 10 can be produced by adding up the mts of the left half and the mps of the right half. However, the problem is not completely solved yet. These auxiliary values should also be calculated during D&C, and the same issue shall occur again: no corresponding combinator exists unless new auxiliary values are introduced. Figure 11 demonstrates such a case, where the mps of the whole list covers the full left half, and its sum cannot be produced using only mps, mts, and mss of the two halves. Here, we can introduce the sum of integers in the list as a supplementary auxiliary value to enable the calculation of mps and mts. In this way, the mps of the whole list in Figure 11 can be produced by adding up the sum of the left half and the mps of the right half.
Decomposition-Based Synthesis for Applying D&C-Like Algorithmic Paradigms

Start symbol \( S \) \( \rightarrow \) \( N \) | \((S,S)\)  
Integer expr \( N \) \( \rightarrow \) \( N + N \) | \( \text{min } N \) | \( \text{max } N \) | \( \text{sum } N \)  
List expr \( N \) \( \rightarrow \) \( \text{Input} \)  

mps \( xs = \text{max}(\sum(xs[:i+1])) \)  
mts \( xs = \text{max}(\sum(xs[i:])) \)  

Fig. 12. The extended program \( L_{aux}^{mss} \) of \( aux \) for the \( mss \) example, where semantics of \( mps \) and \( mts \) are explained using a Python-like syntax.

The task of applying D&C to \( mss \) can be regarded as a lifting problem \( LP(mss, op) \) for operator \( op \) \(((), (xs_L, xs_R)) \xrightarrow{c} xs_L + xs_R\). The raw specification of this lifting problem is as follows.

\[
(mss \triangle aux) (xs_L + xs_R) = \text{comb} ((), (mss \triangle aux)^2 (xs_L, xs_R))
\]

which can be transformed into a more readable form as shown below.

\[
mss' (xs_L + xs_R) = \text{comb} ((), (mss' xs_L, mss' xs_R)), \text{where } mss' \triangleq mss \triangle aux
\]

Fig. 13. The expected \( aux \) and \( comb \) for \( mss \).

4.2 Decomposition System

\( \text{AutoLifter} \) decomposes lifting problems using two decomposition methods, \( \text{component elimination} \) and \( \text{variable elimination} \). For clarity, for each method, we shall first discuss its general idea on a compact specification and then show how to apply it to decompose lifting problems.

4.2.1 Component Elimination. In lifting problems, the output of \( \text{comb} \) is a pair of two components, corresponding to the expected output of the original program and the auxiliary values defined by \( aux \). Our first decomposition method, component elimination, is proposed for decomposing a lifting problem into two subtasks, each involving only one component in the output pair.

\( \text{General idea} \). The task considered by component elimination is to calculate the output of a program from the output of another program. The general specification of this task is shown below.

\[
\forall (x,y) \in S, (\text{out } x, g x) = f (\text{in } y, g y)
\]

In this specification, \( \text{out} \) and \( \text{in} \) are known functions with the same input type, \( S \) is a set of input pairs, each comprising an input of \( \text{out} \) and an input of \( \text{in} \), and \( f \) and \( g \) are two unknown programs to be synthesized. Intuitively, \( g \) specifies some auxiliary values, and \( f \) calculates the output of \( \text{out} \) and the new auxiliary values from the output of \( \text{in} \) and the corresponding auxiliary values.
Component elimination decomposes this task into two subtasks, each involving sub-programs of \( f \) and \( g \). Specifically, since \( f \) is required to output a pair of two values, we can assume the target program of \( f \) is formed by two sub-programs \( f_1 \) and \( f_2 \) such that \( f = f_1 \cdot f_2 \), each for calculating one value in the output pair. Furthermore, if the program space of \( f \) includes operators for accessing tuples, without loss of generality, we can assume \( g \) is formed by two sub-programs \( g_1 \) and \( g_2 \) such that \( g = g_1 \cdot g_2 \), where \( g_1 \) provides the auxiliary values for \( f_1 \), and \( g_2 \) provides the extra auxiliary values needed by \( f_2 \). Then, the specification is decomposed into two subtasks.

1. The first subtask synthesizes \( f_1 \) and \( g_1 \) from the specification below, where \( f_1 \) returns the output of \( out \), and \( g_1 \) provides necessary auxiliary values for \( f_1 \).

\[
\forall (x, y) \in S, \ out \ x = f_1 (in \ y, g_1 \ y)
\]  

(12)

2. If \( g_1 \) is \( null \), i.e., no auxiliary value is needed, a solution for the full specification (Formula 11) is \( f = f_1 \cdot null \) and \( g = null \). At this time, the second subtask is not needed.

3. Otherwise, the second subtask synthesizes \( f_2 \) and \( g_2 \) from the specification below, where \( f_2 \) returns the new auxiliary values and \( g_2 \) extends the resulting \( g_1 \) of the first subtask with extra auxiliary values.

\[
\forall (x, y) \in S, \ (g_1 \ x, g_2 \ x) = f_2 ((in \cdot g_1) \ y, g_2 \ y)
\]  

(13)

The second subtask has the same form as the full specification. Therefore, it can be recursively decomposed by component elimination.

4. Using the results of subtasks, a solution for the full specification can be constructed as follows.

\[
f (v_{in}, (v_1, v_2)) = (f_1 (v_{in}, v_1), f_2 ((v_{in}, v_1), v_2)) \quad \text{g} \ y = (g_1 \ y, g_2 \ y)
\]

Please note that the above decomposition is approximate. There is no guarantee that the auxiliary values found in the first subtask (i.e., the output of \( g_1 \)) can be calculated using programs in the program spaces of \( f_2 \) and \( g_2 \). Consequently, the second subtask may be unrealizable.

**Usage in AutoLifter.** Let us first rewrite the specification of the lifting problem (Formula 9) into the following equivalent form.

\[
(\text{orig} (op (c, \bar{a})), \text{aux} (op (c, \bar{a}))) = (\text{comb} (c (\text{orig}^n \bar{a}, \text{aux}^n \bar{a})))
\]  

(14)

We can see that the above formula has mostly the same form as the general form in Formula 11 (repeated below), where the correspondence is shown in the following table.

<table>
<thead>
<tr>
<th>General Form</th>
<th>Lifting Problem</th>
<th>General From</th>
<th>Current Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( \text{comb} )</td>
<td>\text{in}</td>
<td>\text{orig}^n</td>
</tr>
<tr>
<td>( g )</td>
<td>( \text{aux} )</td>
<td>\text{out}</td>
<td>\text{orig}</td>
</tr>
<tr>
<td>( x )</td>
<td>( \text{op} (c, \bar{a}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( y )</td>
<td>( \bar{a} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The two differences here are that (1) \( \text{comb} \) has an extra parameter \( c \) and (2) \( \text{aux} \) is applied to every element of a tuple. Such differences do not affect the core idea of component elimination, and our general discussion can be trivially extended to cover this form.

Following the decomposition procedure of component elimination, we can assume the form of \( \text{comb} \) as a pair of two sub-programs \( \text{comb}_1 \) and \( \text{comb}_2 \), assume the form of \( \text{aux} \) as a pair of two sub-programs \( \text{aux}_1 \) and \( \text{aux}_2 \), and then decompose the lifting problems as follows.

1. The first subtask synthesizes \( \text{comb}_1 \) and \( \text{aux}_1 \) from the specification below.

\[
\text{orig} (op (c, \bar{a})) = \text{comb}_1 (c, (\text{orig}^n \bar{a}, \text{aux}_1^n \bar{a}))
\]  

(15)
This subtask can still be recursively decomposed by component elimination. At this time, we shall ensure such a function exists, which means, no auxiliary value is needed to calculate the expected output of \( \text{orig} \), a solution for the lifting problem is \( \text{comb} \triangleq \text{comb}_1 \triangleq \text{null} \) and \( \text{aux} \triangleq \text{null} \). At this time, the second subtask is not needed.

(3) Otherwise, the second subtask synthesizes \( \text{comb}_2 \) and \( \text{aux}_2 \) from the specification below, where \( \text{comb}_1 \) and \( \text{aux}_1 \) denote the result of the first subtask. This task can be recursively solved in the same way.

\[
(\text{aux}_1 \triangleq \text{aux}_2) (\text{op} (c, \overline{a})) = \text{comb}_2 (c, ((\text{orig} \triangleq \text{aux}_1)^n \overline{a}, \text{aux}_2^n \overline{a})) \tag{16}
\]

(4) Given the results of the two subtasks, a solution for the lifting problem can be obtained by taking \( \text{aux} \) as \( \text{aux}_1 \triangleq \text{aux}_2 \) and constructing \( \text{comb} \) by pairing up \( \text{comb}_1 \) and \( \text{comb}_2 \) with properly adjusting the structure of their inputs.

**Example 4.1.** After applying component elimination to the \( \text{mss} \) task (Formula 10), the specification of the first subtask is as follows.

\[
\text{mss} (\text{xs}_L \oplus \text{xs}_R) = \text{comb}_1 ((), ((\text{mss} \text{xs}_L, \text{mss} \text{xs}_R), (\text{aux}_1 \text{xs}_L, \text{aux}_1 \text{xs}_R))) \tag{17}
\]

One can verify that components \( a @ \text{mps}, a @ \text{mts} \) and \( c @ \text{mss} \) in Figure 13 form a valid solution for this subtask. Given this solution, the specification of the second subtask is as follows.

\[
(\text{aux}_1 \triangleq \text{aux}_2) (\text{xs}_L \oplus \text{xs}_R) = \text{comb}_2 ((), ((\text{known} \text{xs}_L, \text{known} \text{xs}_R), (\text{aux}_2 \text{xs}_L, \text{aux}_2 \text{xs}_R)))
\]

where \( \text{aux}_1 \triangleq (\text{mps} \triangleq \text{mts}) \) and \( \text{known} \triangleq \text{mss} \triangleq \text{aux}_1 \)

This subtask can be recursively decomposed by component elimination. The first subtask of this recursive decomposition (denoted as subtask 2.1) is as follows.

\[
\text{aux}_1 (\text{xs}_L \oplus \text{xs}_R) = \text{comb}_{2,1} ((), ((\text{known} \text{xs}_L, \text{known} \text{xs}_R), (\text{aux}_{2,1} \text{xs}_L, \text{aux}_{2,1} \text{xs}_R)))
\]

where \( \text{aux}_1 \triangleq (\text{mps} \triangleq \text{mts}) \) and \( \text{known} \triangleq \text{mss} \triangleq \text{aux}_1 \)

One can verify that components \( a @ \text{sum}, c @ \text{mps}, \) and \( c @ \text{mts} \) in Figure 13 form a valid solution for subtask 2.1, which leads to the following subtask 2.2.

\[
(\text{aux}_{2,1} \triangleq \text{aux}_{2,2}) (\text{xs}_L \oplus \text{xs}_R) = \text{comb}_{2,2} ((), ((\text{known} \text{xs}_L, \text{known} \text{xs}_R), (\text{aux}_{2,2} \text{xs}_L, \text{aux}_{2,2} \text{xs}_R)))
\]

where \( \text{aux}_{2,1} \triangleq \text{sum} \) and \( \text{known} \triangleq (\text{mss} \triangleq (\text{mps} \triangleq \text{mts})) \triangleq \text{aux}_{2,1} \)

This subtask can still be recursively decomposed by component elimination. At this time, we shall get \( \text{null} \) for the first sub-program of \( \text{aux}_{2,2} \), representing that no extra auxiliary value is needed, and thus the synthesis finishes.

**4.2.2 Variable Elimination.** The first task of component elimination involves two unknown programs \( \text{aux}_1 \) and \( \text{comb}_1 \) that occur in the form of a composition. Our second decomposition method, variable elimination, is proposed for decomposing this task into two subtasks, each involving only one unknown program.

**General idea.** Variable elimination aims to eliminate an unknown program from an input-output specification. Specifically, it considers specifications in the following form, where \( f \) and \( g \) are two unknown programs to be synthesized.

\[
V(x, y) \in S, x = f (\text{in} g y) \tag{18}
\]

This is an input-output specification for program \( f \), where \( S \) is a set of value pairs, each comprising an output of \( f \) and a parameter for generating the input, and \( \text{in} \) is a second-order program that generates an input of \( f \) using the other unknown program \( g \) and a parameter \( y \).

Variable elimination decomposes this specification using the fact that \( f \) acts as a function. To ensure such a function exists, \( g \) must ensure that for any two pairs in \( S \), the inputs of \( f \) must be different when the expected outputs differ. Using this property, variable elimination decomposes

---

the full specification (Formula 18) into the two sequential subtasks below, where the $g$ synthesized from the first subtask (Formula 19) is put into the second one (Formula 20), making the input and the output of $f$ no longer symbolic in the second subtask.

\[
\forall (x, y), (x', y') \in S, x \neq x' \rightarrow in \ g y \neq in \ g y' \tag{19}
\]

\[
\forall (x, y) \in S, x = f (in \ g y) \tag{20}
\]

Note that the above decomposition is approximate. The resulting $g$ synthesized from the first subtask only ensures that a function exists for $f$ to implement, but there is no guarantee that this function can be implemented within the program space of $f$.

Usage in AutoLifter. The first task generated by component elimination (Formula 15, repeated below) has the same form as the general form (Formula 18, repeated below). We list the correspondence in the table below.

<table>
<thead>
<tr>
<th>General Form</th>
<th>Current Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$comb_1$</td>
</tr>
<tr>
<td>$g$</td>
<td>$aux_1$</td>
</tr>
<tr>
<td>$x$</td>
<td>$orig (op (c, a))$</td>
</tr>
<tr>
<td>$y$</td>
<td>$(c, \overline{a})$</td>
</tr>
<tr>
<td>$in$</td>
<td>$\lambda aux_1, \lambda (c, a). (c, (orig^n a, aux_1^n a))$</td>
</tr>
</tbody>
</table>

Following the decomposition procedure of variable elimination, we can further decompose the first task of component elimination into the following two subtasks.

(1) The first subtask targets synthesizing $aux_1$ from the specification below.

\[
orig (op (c, a)) \neq orig (op (c', a')) \rightarrow (c, (orig^n a, aux_1^n a)) \neq (c', (orig^n a', aux_1^n a'))
\]

We transform this specification into the following equivalent form to clarify the constraint on $aux_1$. Following the notation in Section 2, we include leaf subtasks of decomposition within framed boxes to distinguish them from the other intermediate subtasks.

\[
(orig^n a = orig^n a' \land orig (op (c, a)) \neq orig (op (c, a'))) \rightarrow aux_1^n a' \neq aux_1^n a' \tag{21}
\]

(2) The second subtask targets synthesizing $comb_2$ from the specification below, where $aux_1$ here is the program obtained by solving the first subtask.

\[
orig (op (c, a)) = comb_1 (c, (orig^n a, aux_1^n a)) \tag{22}
\]

Example 4.2. After applying variable elimination to the first subtask generated in the previous example (Formula 17), the specification of the first subtask is shown below.

\[
(mss xs_L, mss xs_R) = (mss xs'_L, mss xs'_R) \land mss (xs_L + xs_R) \neq mss (xs'_L + xs'_R) \\
\rightarrow (aux_1 xs_L, aux_1 xs_R) \neq (aux_1 xs'_L, aux_1 xs'_R)
\]

One can verify that components $a@mps$ and $a@mts$ form a valid solution to this subtask.
4.2.3 Decomposition System. As mentioned before, two strategies exist for applying the decomposition methods, greedy and backtracking. Here we focus on the greedy strategy, the one used in our implementation. Algorithm 1 shows the pseudocode of a decomposition system using the greedy strategy.

1. Algorithm 1 starts the decomposition by applying component elimination to the original lifting problem (Line 13). The first subtask is further decomposed by variable elimination (Lines 7-8), and the second one is solved by recursively applying component elimination (Lines 10-11). The recursion terminates when no new auxiliary value is found (Line 9).

2. Algorithm 1 applies variable elimination to the first subtask generated by component elimination (Lines 1-5) and solves the two subtasks via inductive synthesis (Lines 3 and 5), which shall be discussed later (Section 4.3).

Example 4.3. Figure 14 illustrates how Algorithm 1 decomposes the mss task, where nodes are subtasks (#1 denotes the original problem), arrows indicate task decomposition, tags within nodes indicate the sub-program synthesized from each subtask, and different line styles indicate different decomposition/subtask types. As we can see, the scale of the leaf subtasks (#3, #4, #7, #8, #11, and #12, each including at most 2 components) is greatly reduced compared to the original lifting problem (#1, including 7 components). There are four types of subtasks: (1) lifting problems, including the original lifting problem (#1) and the second subtasks of component elimination (#5, #9), (2) the first subtasks of component elimination (#2, #6, #10), (3) the first subtasks of variable elimination (#3, #7, #11), which involve only sub-programs aux, and (4) the second subtasks of variable elimination (#4, #8, #12), which involve only sub-programs of comb.

Besides, the following points are worth noting about Figure 14.

- The tags in each node represent the synthesis result of the subtask. They are unavailable when the subtask is generated or further decomposed.
- For both decomposition methods, the second subtask relies on the synthesis result of the first one so that it will not be generated until the first subtask is solved. The indices of nodes in Figure 14 reflect the generation order of subtasks.
- The decomposition terminates after solving subtask #10 (a first subtask of component elimination) because the synthesis result shows that no new auxiliary value is required.
4.3 Inductive Synthesis for Leaf Tasks

The decomposition system generates two types of leaf tasks, corresponding to the two subtasks of variable elimination (Formulas 21 and 22), respectively. We apply the CEGIS framework [Solar-Lezama et al. 2006] to convert both types of tasks into example-based synthesis tasks.

CEGIS (Algorithm 2) synthesizes by iteratively invoking an example-based synthesizer and a verifier. It records an example set that is initially empty (Line 1). In each iteration, the example-based synthesizer generates a candidate program $\text{prog}$ from existing examples (Line 3) and the verifier generates a counter-example $e$ under which $\text{prog}$ is incorrect, i.e., $\neg \phi(\text{prog}, e)$ is satisfied (Line 4). The candidate program will be returned if it is verified to be correct (i.e., no counter-example exists) (Line 5). Otherwise, the counter-example will be recorded for further synthesis (Line 6).

In this paper, we assume the existence of the verifier for both types of leaf tasks and focus on the example-based synthesis tasks. In practice, the verifier can be selected among off-the-shelf ones on demand. In our implementation, we use a combination of bounded model checking [Biere et al. 2003] and random testing by default.

**Example-based synthesizer for $\text{comb}$**. We begin with the leaf subtask of $\text{comb}$ (Formula 22), the simpler case. An example of this task is an assignment to $(c, \overline{a})$ and is in the input-output form, requiring $\text{comb}_1$ to output $\text{orig}(\text{op}(c, \overline{a}))$ from input $(c, (\text{orig} \circ \text{aux}_1)^n \overline{a})$. This task can be solved by those existing synthesis algorithms relying on input-output examples. As shown in Algorithm 3, the example-based synthesizer for $\text{comb}$ just converts the given examples into the input-output form (Lines 2-6) and then passes them to existing synthesizers.

**Example-based synthesizer of $\text{aux}$**. For the leaf subtask of $\text{aux}$ (Formula 21), an example is an assignment to $(c, \overline{a}, \overline{a})$, with a constraint of $\text{aux}_1^n \overline{a} \neq \text{aux}_1^n \overline{a}$. Note that the premise of this specification can be ignored in the example-based task because any example generated by CEGIS must be a counter-example of some candidate program, which is only possible when the premise of the specification is true.
The maximum prefix sum mps satisfies example $\overline{x}_3$ because $mps$ generates two different outputs $(1, 1)$ and $(1, 0)$ on $([1], [1])$ and $([1], [-1, 1])$. Similarly, the maximum tail-segment sum mts satisfies example $\overline{x}_2$, and their pair $mts \odot mps$ satisfies both examples.

The example-based synthesizer for aux (Algorithm 4) is built upon an existing enumerative synthesizer, namely observational equivalence (OE) [Udupa et al. 2013] (Line 10). OE enumerates programs from small to large following a bottom-up manner. It constructs larger programs by combining those existing smaller programs via language constructs. OE uses an effective pruning strategy to avoid duplicated programs with the same input-output behaviors. This strategy is parameterized by an input set and will prune off those programs producing duplicated outputs on this input set (compared to existing programs).

Example 4.5. Consider a synthesis task with a single input $x$. When the input set is $\{1\}$, OE will skip program $x \times 2$ if $x + 1$ has been visited before because both programs output 2 from the input. Furthermore, those programs constructed from $x \times 2$, such as $(x \times 2) + 1$ and $(x \times 2) \times x$, will be implicitly skipped as well because $x \times 2$ will no longer be used to construct larger programs.

In the example-based synthesis task of aux, whether a program satisfies an example $(c, \overline{a}, \overline{a'})$ is determined by its outputs on those components inside $\overline{a}$ and $\overline{a'}$. Therefore, OE can be applied by including all inputs involved in examples into the input set (Lines 1-2, Algorithm 4).

Example 4.6. When the example set is $\{\overline{x}_1, \overline{x}_2\}$ (examples in Example 4.4), OE can be invoked with input set $\{[1], [-1,1], [1,-1]\}$. Any two programs outputting the same from these inputs must satisfy the same subset of examples.

Optimization: observational covering. The example-based synthesizer for aux also includes a specialized optimization for better handling the case where multiple auxiliary values are required. Specifically, we observe that many practical tasks require multiple auxiliary values, for example,
\begin{algorithm}
\textbf{Input}: An example set \textit{examples} and an integer \textit{lim}_c specifying the number of components considered by observational covering.
\textbf{Output}: A valid auxiliary program \textit{aux}_1.
\begin{algorithmic}[1]
1 \textbf{involvedInputs} $\leftarrow \{a \mid (c, \overline{a}, a') \in \textit{examples} \land (a \in \overline{a} \lor a \in a')\}$;
2 \textbf{oe} $\leftarrow$ \text{ObservationalEquivalenceSolver} (\textit{involvedInputs});
3 \textbf{size} $\geq 0$, \textit{programs}[\textit{size}] $\leftarrow [\ ]$; \textbf{result} $\leftarrow \bot$;
4 \textbf{Function} \text{IsCovered}(\textit{prog}, \textit{size})$:\n\hspace*{2em} return $\exists \textit{size}' \leq \textit{size}, \exists \textit{prog}' \in \textit{programs}[\textit{size'}], \textit{prog}'$ satisfies all examples that are satisfied by \textit{prog};
5 \textbf{Function} \text{Insert}(\textit{prog}, \textit{size})$:\n\hspace*{2em} if \textit{prog} satisfies all examples $\land \textbf{result} = \bot$ then \textbf{result} $\leftarrow \textit{prog}$;
6 \hspace*{2em} if $\neg$\text{IsCovered}(\textit{prog}, \textit{size}) then \textit{programs}[\textit{size}].\text{Append}(\textit{prog});
7 \textbf{Function} \text{Extend}()$:\n\hspace*{2em} \textbf{component} $\leftarrow$ \textbf{oe}.\text{Next}(); \text{Insert}(\textbf{component}, 1)
8 \hspace*{2em} \textbf{prePrograms} $\leftarrow$ \textbf{programs};
9 \hspace*{2em} \textbf{foreach} \textit{size} $\in [1, \ldots, \text{\textit{lim}_c} - 1]$ and \textit{prog} $\in \text{\textbf{prePrograms}[\textit{size}]}$ do
10 \hspace*{3em} \text{Insert}(\textit{prog} $\oplus$ component, size + 1);
11 \hspace*{2em} \textbf{end}
12 \hspace*{2em} \text{Insert}(\text{null}, 0);
13 \hspace*{2em} while \textbf{result} $= \bot$ \text{do} \text{Extend}();
14 return \textbf{result};
\end{algorithmic}
\end{algorithm}

\textit{mps} and \textit{mts} are both required for calculating \textit{mss} in D&C. In these cases, the form of \textit{aux}_1 can be assumed as a tuple of components (i.e., \textit{comp}_1 $\wedge \ldots \wedge \textit{comp}_c$), each in a smaller scale. To better synthesize such programs, we optimize the enumeration by invoking OE to generate only basic components and combining these components explicitly on the top level (Lines 6-14).

To implement the optimized enumeration, our example-based synthesizer for \textit{aux} maintains a program storage \textit{programs} during the enumeration, where \textit{programs}[\textit{size}] stores existing programs formed by \textit{size} components (Line 3). In each iteration, our synthesizer invokes OE to generate the next component (Line 10) and then combines it with existing programs to form larger tuples (Lines 11-14). To limit the combination space, our synthesizer is configured by an integer \textit{lim}_c and considers only combining at most \textit{lim}_c components at the top level (Line 12). Note that such a limitation would not affect the effectiveness: our synthesizer is still complete (i.e., never fails on a realizable task) even when \textit{lim}_c is set to 1. A valid program of a realizable task will ultimately be found as a single component because OE directly enumerates the whole program space.

To further speed up the combination, we propose an optimization method named \textit{observational covering}. This method follows the key idea of OE, that is, to prune off programs whose effect is covered by some other programs on a set of given examples. Recall that an example $(c, \overline{a}, a')$ here requires the auxiliary program to return different results on \overline{a} and \overline{a'} which means, an example is satisfied by an auxiliary program when and only when it is satisfied by some components in the auxiliary program. Therefore, when the goal is to satisfy a set of given examples using at most \textit{lim}_c components, the effect of a program \textit{prog} is covered by another program \textit{prog}' if (1) \textit{prog}' uses fewer components than \textit{prog} and (2) \textit{prog}' satisfies all examples satisfied by \textit{prog}. At this time, the covered program \textit{prog} can be safely skipped from the top-level combination.

\textit{Example 4.7}. When the example set is \{$\overline{x}_3$, \overline{x}_2$} (Example 4.4), the effect of \textit{max} is covered by \textit{null} (the empty auxiliary program) because both programs satisfy no example and \textit{max} uses one
more component. Therefore, \( \text{max} \) can be safely skipped from the combination: whenever there is a program combined from \( \text{max} \) (assumed as \( \text{prog} \wedge \text{max} \)) satisfying both examples, there must exist another valid program \( \text{prog} \wedge \text{null} \) (i.e., \( \text{prog} \)) using fewer components.

Using this property, our synthesizer skips all covered programs. Only programs that are not covered by existing ones will be inserted into the storage for further combination (Lines 4-5 and 8).

**Example 4.8.** Algorithm 4 runs as below when the example set is \( \{ \overline{x_1}, \overline{x_2} \} \) (Example 4.4), the limit \( \lim_c \) is 2, and the first three components returned by OE are \( \text{max} \), \( \text{mps} \), and \( \text{mts} \) in order.

- Before the first invocation of \( \text{Extend} \), \( \text{null} \) is inserted and the storage is \( \text{programs}[0] = [\text{null}] \).
- In the first invocation, OE generates \( \text{max} \) and no other program is constructed. \( \text{max} \) will not be inserted into the storage as it is covered by \( \text{null} \). So the storage will remain unchanged.
- In the second invocation, OE generates \( \text{mps} \) and no other program is constructed. \( \text{mps} \) is not covered by \( \text{null} \) as it satisfies \( \overline{x_1} \), an example violated by \( \text{null} \). So \( \text{mps} \) will be inserted, and the storage will become \( \text{programs}[0] = [\text{null}], \text{programs}[1] = [\text{mps}] \).
- In the third invocation, OE generates \( \text{mts} \) and program \( \text{mps} \wedge \text{mts} \) is generated by combination. Both programs are not covered and will be inserted, and the storage will become as follows.

\[
\text{programs}[0] = [\text{null}] \quad \text{programs}[1] = [\text{mps}, \text{mts}] \quad \text{programs}[2] = [\text{mps} \wedge \text{mts}]
\]

Then \( \text{mps} \wedge \text{mts} \) will be returned as the result as it already satisfies all given examples.

## 5 PROPERTIES

### 5.1 Soundness

**AutoLifter** is sound when the verifiers of leaf subtasks are sound (Theorem 5.1). Specifically, when these verifiers are sound, the synthesis results of the leaf subtasks must satisfy their respective specifications. Recall that in every decomposition made by **AutoLifter**, the second subtask is always for completing the synthesis result of the first subtask into a valid solution for the original task. Therefore, the final results built up from correct results of leaf subtasks must also be correct, implying the soundness of **AutoLifter**.

**Theorem 5.1 (Soundness).** The result of **AutoLifter** (Algorithm 1) is valid for the original lifting problem if the verifiers of leaf subtasks accept only valid programs for respective subtasks.

**Proof.** Proofs of the lemmas and theorems in this paper are available in Appendix A.1. \( \square \)

### 5.2 Cost of Approximation

Recall that **AutoLifter** uses approximate specifications when decomposing lifting problems. Such a treatment, in theory, will have negative effects on performance. The form of such effects depends on the specific strategy used to implement the decomposition system.

- For the greedy strategy, every unrealizable subtask will make **AutoLifter** fail in solving lifting problems, affecting the effectiveness.
- For the backtracking strategy, every unrealizable subtask will make **AutoLifter** roll back and search for other solutions, affecting the efficiency.

Fortunately, we have empirical results showing that the negative effects of our approximations are negligible. In our evaluation, **AutoLifter** never decomposes realizable tasks into unrealizable and can solve almost all realizable tasks within a short time (Section 8.2).

The direct reason for this phenomenon is the excellent practical performance of our example-based synthesizer for \( \text{aux} \) (Algorithm 4). Specifically, each subtask generated by **AutoLifter** depends only on the original lifting problem and the sub-programs of \( \text{aux} \) synthesized previously, making...
the decomposition procedure of AutoLifter fully determined by the result of the aux synthesizer. Our aux synthesizer works well on this aspect. In our evaluation, it can always find the intended auxiliary program and thus avoid AutoLifter from generating unrealizable subtasks.

As discussed in the snadmin example (Section 2.4), we ascribe the effectiveness of our aux synthesizer to two reasons: (1) the compressing property of a practical lifting problem (i.e., both orig and programs in \( L_{aux} \) map a large input space to a small output space) makes the specification of aux (Formula 21) strong enough to exclude most candidates, and (2) the preference to simpler auxiliary programs helps avoid those unnecessarily complex solutions. In the remainder of this section, we shall provide formal results corresponding to these two factors.

- First, in Section 5.2.1, we analyze the probability for AutoLifter to generate unrealizable subtasks under a probabilistic model, where the semantics of \( L_{aux} \) and \( L_{comb} \) are modeled as random. Then in Section 5.2.2, we prove that this probability is almost surely small under the compressing property (Theorem 5.6).
- Second, in Section 5.2.3, we consider the concrete semantics of \( L_{aux} \) and \( L_{comb} \) and prove that our aux synthesizer (Algorithm 4) can always find a minimal solution (Theorem 5.8).

### 5.2.1 Probability to generate unrealizable subtasks

To study the effectiveness of AutoLifter, we aim to analyze the probability for AutoLifter to generate unrealizable subtasks. However, calculating this probability precisely is extremely difficult because the languages of candidate programs (\( L_{aux} \) and \( L_{comb} \)) are usually complex. Both of these languages may include infinitely many candidate programs, each assigned with possibly complex semantics. Consequently, it is almost impossible to precisely predict the performance of AutoLifter in synthesis.

We overcome this challenge by introducing a probabilistic model and conducting an approximate analysis instead. Specifically, we assume the semantics of programs in \( L_{aux} \) and \( L_{comb} \) as random functions and then analyze the probability for AutoLifter to generate unrealizable subtasks.

The following are the details of our probabilistic analysis.

Given lifting problem LP\((\text{orig}, \text{op})\), we construct a corresponding probabilistic model \( M[\text{orig}, \text{op}] \) by modeling the semantics of programs in \( L_{aux} \) and \( L_{comb} \) as uniformly random functions. The detailed construction of \( M[\text{orig}, \text{op}] \) is shown below.

- \( M[\text{orig}, \text{op}] \) is constructed on a set of given parameters, including (1) the types and semantics of \( \text{orig} \) and \( \text{op} \), and (2) the syntax and the type of every program in \( L_{aux} \) and \( L_{comb} \). The only thing this model does is to assign random semantics to programs in \( L_{aux} \) and \( L_{comb} \).
- For simplicity, we make the following assumptions when constructing \( M[\text{orig}, \text{op}] \):
  1. Programs in \( L_{aux} \) are formed as tuples of components (i.e., \( \text{comp}_1 \triangleq \ldots \triangleq \text{comp}_k \)), where each component \( \text{comp}_i \) outputs only a single auxiliary value.
  2. There is a universal value type \( V \) capturing the types of the output of \( \text{orig} \), every auxiliary value, and the complementary input required by \( \text{op} \).
  3. The numbers of different values in the input type \( A \) (the input type of \( \text{orig} \)) and the value type \( V \) are both finite\(^3\), denoted as \( s_A \) and \( s_V \), respectively. At this time, the compressing property can be modeled as a domain property that \( s_A \gg s_V \).
- \( M[\text{orig}, \text{op}] \) generates a lifting problem by independently and uniformly sampling the semantics for every candidate program in \( L_{aux} \) and \( L_{comb} \). For example, suppose that there is a program in \( L_{aux} \) with type \( A \to V \). The semantics of this program will be uniformly drawn from functions mapping from \( A \) to \( V \) (\( s_V^A \) possibilities in total).

\(^3\)Although there exist types including infinitely many values (e.g., lists), they can be approximated in our model by taking a large enough finite subset (e.g., setting a large enough length limit for lists).
The main result of our analysis is to bound the size-limited unrealizable rate of AutoLifter under the probabilistic model using the mismatch factor of the given lifting problem. Before going into the details of this result, we shall first introduce the two concepts used in the analysis, starting from the size-limited unrealizable rate (Definition 5.2).

Definition 5.2 (Size-Limited Unrealizable Rate). Given a lifting problem \( \text{LP}(\text{orig}, \text{op}) \) and an integer \( \lim_s \), the size-limited unrealizable rate of AutoLifter, denoted as \( \text{unreal}(\text{orig}, \text{op}, \lim_s) \), is defined as the probability for AutoLifter to decompose a random lifting problem (sampled from \( M[\text{orig}, \text{op}] \)) into unrealizable subtasks, under the condition that the random problem has a valid solution whose size is no larger than \( \lim_s \), i.e.,

\[
\Pr_{\phi \sim M[\text{orig}, \text{op}]} \left[ \text{AutoLifter generates an unrealizable subtask from } \phi \right] \\
\exists (\text{aux}, \text{comb}), \left( \text{size}(\text{aux}, \text{comb}) \leq \lim_s \land (\text{aux}, \text{comb}) \text{ is valid for } \phi \right)
\]

where \( \phi \sim M[\text{orig}, \text{op}] \) represents that \( \phi \) is a random lifting problem sampled from \( M[\text{orig}, \text{op}] \) and \( \text{size}(\text{aux}, \text{comb}) \) represents the total size of programs \( \text{aux} \) and \( \text{comb} \).

In this definition, we take the size of the smallest valid program of lifting problems into consideration by introducing the size limit \( \lim_s \). Such a treatment enables a more refined analysis that relies on the size of the smallest valid program.

The second concept is the mismatch factor of a lifting problem \( \text{LP}(\text{orig}, \text{op}) \) (Definition 5.3). This factor counts the number of independent examples that \( \text{aux} \) must satisfy to ensure a function exists for calculating the output of \( \text{orig} \).

Definition 5.3 (Mismatch Factor). Given lifting problem \( \text{LP}(\text{orig}, \text{op}) \) and integer \( t \), the mismatch factor of this lifting problem is at least \( t \) if there exists \( t \) pairs of values \( (\text{orig}, \text{op}) \in A^n \times A^n \) satisfying the following two conditions.

- For each pair \( (\text{orig}, \text{op}) \), the inputs of \( \text{comb} \) when no auxiliary value is used (i.e., the output of \( \text{orig} \)) are the same but the outputs of \( \text{comb} \) (i.e., the output of \( \text{orig} \circ \text{op} \)) are different. In other words, every pair \( (\text{orig}, \text{op}) \) must satisfy the formula below.

\[
\exists c \in C, \left( \text{orig}^n \text{orig}, \text{op} \neq \text{orig}^n \text{op} \right) \land (\text{orig} \circ \text{op})(c, \text{orig}) \neq (\text{orig} \circ \text{op})(c, \text{op}) \right)
\]

- All components involved in these pairs (2nt in total) are different.

Recall that \( A \) denotes the input type of \( \text{orig} \), \( n \) denotes the arity of \( \text{op} \), and \( C \) denotes the type of the complementary input required by \( \text{op} \).

The mismatch factor reflects the strength of the specification that AutoLifter uses to synthesize \( \text{aux} \). To see this point, let us consider the first leaf subtask of \( \text{aux} \) (Formula 21) generated when solving lifting problem \( \text{LP}(\text{orig}, \text{op}) \), as shown below.

\[
(\text{orig}^n \text{orig} = \text{orig}^n \text{op} \neq \text{orig}^n \text{op} \circ \text{c} \text{orig}) \rightarrow \text{aux}^n \text{aux} \neq \text{aux}^n \text{op} \circ \text{c} \text{op}
\]

The premise of this specification is exactly the first condition in Definition 5.3. Through this connection, a mismatch factor of at least \( t \) implies that, in the approximate specification derived by AutoLifter, \( \text{aux} \) (a sub-program of \( \text{aux} \)) needs to output differently on \( t \) independent pairs of inputs. Intuitively, with a larger mismatch factor, an incorrect \( \text{aux} \) will be less likely to satisfy this specification, and thus AutoLifter will be more likely to solve the lifting problem successfully.

With the above two concepts, Theorem 5.4 shows the main result of our probabilistic analysis. We prove that for any lifting problem, the size-limited unrealizable rate of AutoLifter is bounded by the mismatch factor, the size limit, and the size of the output domain (i.e., \( s_V \)).
Theorem 5.4 (Upper Bound on the Unrealizable Rate). Given a lifting problem LP\((\text{orig}, \text{op})\) of which the mismatch factor is at least \(t\), the size-limited unrealizable rate of AutoLifter is bounded, as shown below.

\[
\text{unreal}(\text{orig}, \text{op}, \lim_s) \leq 2^w \exp\left(-t/s_V^w\right), \text{ where } w \triangleq \lim_c \cdot \lim_s
\]

5.2.2 Effectiveness under the compressing property. Theorem 5.4 shows that the unrealizable rate of AutoLifter is small when the mismatch factor is far larger than the size of the output domain. Fortunately, this is the usual case when solving lifting problems. One important domain property here is the compressing property of lifting problems, that is, the input domain of \(\text{orig}\) and auxiliary programs in \(\mathcal{L}_{\text{aux}}\) are usually far larger than their output domains. In the following, we shall first introduce the reason why the compressing property generally exists in lifting problems and then discuss how the compressing property implies a large mismatch factor.

To see why the compressing property exists, let us recall the meaning of lifting problems discussed in Section 3.2. In the sense of optimization, the inputs of \(\text{orig}\) and \(\text{aux}\) correspond to the intermediate data structures constructed in an inefficient program, and their outputs correspond to the values calculated after eliminating these intermediate data structures. To achieve optimization, these programs must summarize a small result (e.g., a scalar value) from a large data structure (e.g., an inductive data structure that can be arbitrarily large), leading to the compressing property.

Example 5.5. In the two lifting problems discussed in Sections 2.1 and 3.1 (Formulas 1 and 8), both the original program \(\text{snd\_min}\) and the auxiliary program \(\text{aux}\) take an integer list as the input and output a single integer. The ratio between the number of integer lists to the number of integers tends to \(\infty\) when the length of lists tends to \(\infty\) and every integer is bounded within a fixed range.

To see the relation between the compressing property and the mismatch factor, let us review the first condition in the definition of the mismatch factor (Formula 23). For an input pair \((\overline{a}, \overline{a'})\) and a fixed complementary input \(c\), this condition requires \(\text{orig}^a\) (whose output domain is \(V^n\)) to output the same on \(\overline{a}\) and \(\overline{a'}\), and requires \(\text{orig} \circ \text{op}\) (whose output domain is \(V\)) to output differently on \((c, \overline{a})\) and \((c, \overline{a'})\). The strength of this condition can be estimated using probabilities. The probability for a given \((\overline{a}, \overline{a'})\) and \(c\) to satisfy this condition can be estimated as \(s_V^n(1 - 1/s_V)\) because (1) the probability for two random values in \(V^n\) to be the same is \(s_V^{-n}\) and (2) the probability for two random values in \(V\) to be different is \(1 - 1/s_V\). It means, in the random sense, there is one pair \((\overline{a}, \overline{a'})\) satisfying this condition in every \(O(s_V^n)\) pairs\(^4\). Since there are up to \(S_{\mathcal{A}}/(2n)\) independent pairs in \(A^n \times A^n\) and \(s_A\) is far larger than \(s_V\) by the compressing property, this estimation tells that the mismatch factor is far larger than \(s_V\) with a high probability. A concrete example of analyzing the mismatch factor for a given lifting problem can be found in Appendix A.2.

Theorem 5.6 combines the above analysis with Theorem 5.4. It demonstrates that the unrealizable rate of AutoLifter is almost always small when the compressing property holds.

Theorem 5.6 (Unrealizable Rate Under the Compressing Property). Consider the size-limited unrealizable rate of AutoLifter on a random lifting problem. When there are at least two values (i.e., \(s_V > 1\)), for any constant \(\epsilon > 0\), the probability for this rate to exceed \(\epsilon\) tends to 0 when \(s_{\mathcal{A}}/s_V^w\) tends to \(\infty\), where \(w' \triangleq n \cdot \lim_c \cdot \lim_s + n + 1\).

5.2.3 Preference of AutoLifter for simpler auxiliary programs. In the discussion above, we model the semantics of candidate programs in \(\mathcal{L}_{\text{aux}}\) and \(\mathcal{L}_{\text{comb}}\) as independent to ease the analysis. However, a semantical dependency between programs indeed exists in practice because the semantics is usually defined along with the syntax under domain theories. Such a dependency weakens the

\(^4\)Here we assume that \(s_V > 1\), that means, there are at least two different outputs.
specification for aux (Formula 21) because it makes the realizability of an aux subtask (i.e., the existence of a valid auxiliary program) imply the validness of (infinitely) many other programs.

Example 5.7. In the sndmin example (Section 2.4), many valid programs for the aux subtask (Formula 6) can be constructed from the target auxiliary program min xs, for example, by including more auxiliary values (e.g., \((\text{min } x, \text{max } x)\)) or performing invertible arithmetic operations (e.g., \((\text{min } x) + (\text{min } x))\). Many of them may lead to unrealizable subtasks, for example, the combination function corresponding to \((\text{min } x) + (\text{min } x)\) cannot be implemented in \(L_{\text{comb}}^{\text{ex}}\) (Figure 5b).

The key for AutoLifter to perform well under such a dependency is its preference for simpler auxiliary programs, following the principle of Occam’s razor. The aux synthesizer in AutoLifter (Algorithm 4) is an enumeration-based synthesizer and always returns a minimal possible auxiliary program (Theorem 5.8). Therefore, this synthesizer can successfully avoid those unnecessarily complex programs derived from the expected one.

Theorem 5.8 (Minimality). Given an example-based synthesis task for the auxiliary program, the program aux∗ synthesized by our aux synthesizer must be a minimal valid program. In other words, any strict sub-program of aux∗ must not be valid for the given task.

6 APPLICATIONS OF AUTOLIFTER

We have seen how to reduce the applications of D&C and incrementalization to lifting problems (Sections 2.1 and 4.1). Through these reductions, AutoLifter can be instantiated to synthesizers for the corresponding paradigms. In this section, we shall supply more details on how the efficiency of the resulting program is ensured in these reductions (Section 6.1) and provide reductions for several other D&C-like paradigms (Section 6.2).

6.1 Efficiency Condition

When applying an algorithmic paradigm, the synthesizer needs to ensure that the resulting program must be efficient. We achieve this guarantee by limiting the domain-specific languages \(L_{\text{aux}}\) and \(L_{\text{comb}}\), following the SyGuS framework. Specifically, we require these languages to include only programs that can lead to an efficient result. When instantiating AutoLifter for a specific algorithmic paradigm, we can establish different efficiency guarantees by using different languages.

In this paper, we consider a specific condition (denoted as the efficiency condition) for the domain-specific languages. This condition requires that (1) every program in \(L_{\text{aux}}\) runs in constant time on a constant-sized input, and (2) every program in \(L_{\text{comb}}\) runs in constant time.

Example 6.1. The domain-specific languages \(L_{\text{aux}}^{\text{ex}}\) and \(L_{\text{comb}}^{\text{ex}}\) (Figures 5a and 5b, discussed in the sndmin example) satisfy the efficiency condition.

The efficiency condition implies the efficiency of the D&C and the incremental programs synthesized by the corresponding instantiation of AutoLifter, as shown below.

- In the D&C program synthesized through the reduction in Section 2.1, each of orig, aux, and comb is invoked \(O(n)\) times, and the former two are only invoked on singleton lists. Therefore, this program is ensured to be \(O(n/p)\) time in parallel on a list of length \(n\) and \(p \leq n/\log n\) processors when (1) the efficiency condition is satisfied and (2) orig runs in constant time on a singleton list.
- In the incremental program synthesized through the reduction in Section 3.1, comb will be invoked once after each change. Therefore, this program is ensured to be constant-time per change when the efficiency condition is satisfied.
6.2 More Applications

**Single-Pass.** Single-pass [Schweikardt 2009] is an algorithmic paradigm widely applied in various domains such as databases and networking. It is also the input format required by Parsynt [Farzan and Nicolet 2017, 2021b] (Section 2.1). A single-pass program (Figure 15) scans the input list once from the first element to the last and iteratively updates the result after visiting each element. To apply single-pass to an original program \( \text{orig} \), an auxiliary program \( \text{aux} \) and a combinator \( \text{comb} \) satisfying the formula below are required.

\[
\text{orig}'(\text{xs} + [+v]) = \text{comb}(v, \text{orig}'\text{xs}), \text{where } \text{orig}' \triangleq \text{orig} \triangleaux
\]

This task is equivalent to that for incrementalization (Section 4.1) and can be regarded as a lifting problem \( \text{LP}(\text{orig}, \text{op}) \) with \( \text{op}(v, (\text{xs})) \triangleq \text{xs} + [+v] \). Under the efficiency condition, the resulting single-pass program runs in \( O(n) \) time on a list of length \( n \) because its bottleneck is the \( O(n) \) invocations of \( \text{comb} \) in the loop.

**General Incrementalization.** In the previous incrementalization example (Section 4.1), the only allowed change is to append an element to the back of the input list. In the general case, there can be different types of changes in a single task, captured by a change set \( C \) denoting all possible changes and a change operator \( \text{change} : C \times A \rightarrow A \) applying a change to an \( A \)-element. To incrementally update the result of the original program \( \text{orig} \) after each change, an auxiliary program \( \text{aux} \) and a combinator \( \text{comb} \) satisfying the formula below are required.

\[
\text{orig}'(\text{change}(c, a)) = \text{comb}(c, \text{orig}'a), \text{where } \text{orig}' \triangleq \text{orig} \triangleaux
\]

This task can be regarded as \( \text{LP}(\text{orig}, \text{op}) \) with \( \text{op}(c, (a)) \triangleq \text{change}(c, a) \). Under the efficiency condition, the resulting incremental program must run in \( O(1) \) time per change because only \( \text{comb} \) is invoked once after each change.

**Example 6.2.** Continuing with the previous incrementalization task (Section 3.1), let us consider a new change that pushes a new element to the front of the input list. In the new task, the change set \( C \) can be defined as \( \{ \text{“front”, “back”} \} \times \text{Int} \) and the corresponding change is as follows.

\[
\text{change}((\text{tag}, v), \text{xs}) \triangleq \text{if tag = “front” then } [v] \text{ else } \text{xs} + [+v]
\]

**Longest Segment Problem.** Given a predicate \( p \) and an input list, there may be multiple segments of the input list satisfying \( p \), and the longest segment problem asks for the maximum length of valid segments. Zantema [1992] studies three different subclasses of longest-segment problems and proposes three algorithmic paradigms for them, respectively. Here, we select and introduce one typical paradigm among them, and the details on the others can be found in Appendix B.

This paradigm enumerates segments via a technique named *sliding window* (Figure 16), where \( l \) and \( r \) are the indices of the current segment (from the \( l \)-th to the \( r \)-th element in the input list \( \text{xs} \) ).
and info records (1) whether \( p \) is currently satisfied and (2) necessary auxiliary values. The outer loop appends every element in the input list \( xs \) to the back of the current segment one by one, and the inner loop repeatedly removes the first element of the current segment until \( p \) is satisfied. This enumeration guarantees to visit the longest valid segment when \( p \) is prefix-closed, that is, every prefix of a list satisfying \( p \) satisfies \( p \) as well.

To apply this paradigm to a longest segment problem, an auxiliary program \( aux \) and two combinatorators \( comb_1 \) and \( comb_2 \) are required to correctly update info during the enumeration. Concretely, they must satisfy the formulas below, where \( head \) returns the first element of a list and \( tail \) returns the result of removing the first element from a list.

\[
\begin{align*}
(p \triangleaux) \ (\mathit{xs} \mathbin{\oplus} [v]) &= \mathit{comb}_1 \ (v, (p \triangleaux) \ \mathit{xs}) \\
\text{head } \mathit{xs} &= v \rightarrow (p \triangleaux) \ (\mathit{tail} \ \mathit{xs}) &= \mathit{comb}_2 \ (v, (p \triangleaux) \ \mathit{xs})
\end{align*}
\]

The condition of \( \text{head } \mathit{xs} = v \) is involved to allow the second combinator \( \mathit{comb}_2 \) to access the element to be removed. When reducing the above task to a lifting problem, this condition can be eliminated by assigning a dummy output to \( op \) when the condition is violated, and the two formulas can be merged through the construction in Example 6.2. A possible operator \( op \) of the resulting lifting problem is shown below, where the complementary input is in \{"append", "remove"\} \times Int.

\[
\begin{align*}
op \ ((\mathit{tag}, v), \ (\mathit{xs})) &\triangleq \begin{cases} 
\mathit{xs} \mathbin{\oplus} [v] & \text{tag = "append"} \\
\mathit{tail} \ \mathit{xs} & \text{tag = "remove" } \land \text{head } \mathit{xs} = v \\
\mathit{[]} & \text{otherwise}
\end{cases}
\end{align*}
\]

Under the efficiency condition, the resulting program runs in \( O(n) \) time on a list of length \( n \) because both \( \mathit{aux}_1 \) and \( \mathit{aux}_2 \) are invoked at most \( n \) times.

**Segment Trees.** The segment tree is a type of classical data structure for answering queries on a specific property of a segment in a possibly long list [Bentley 1977b]. Given an initial list, after a linear-time pre-processing, a segment tree can efficiently evaluate a pre-defined function \( \mathit{orig} \) on a segment (e.g., “answer the second minimum of the segment from the 2nd to the 5,000th element”) or applies a pre-defined change operator \( \mathit{change} \) to a segment (e.g., “add each element in the segment from the 2nd to the 5,000th element by 1”), each in logarithmic time w.r.t. the list length.

The detailed template of a segment tree can be found in Appendix B, and in brief, it uses D&C to respond to queries and uses incrementalization to respond to changes. Therefore, implementing a segment tree is to find an auxiliary program \( \mathit{aux} \) and two combinatorators \( \mathit{comb}_1 \) and \( \mathit{comb}_2 \) such that \( (\mathit{aux}, \mathit{comb}_1) \) is a solution to the lifting problem of D&C and \( (\mathit{aux}, \mathit{comb}_2) \) is a solution to the lifting problem of incrementalization; in other words, the formulas below need to be satisfied.

\[
\begin{align*}
(\mathit{orig} \triangleaux) \ (\mathit{xs}_L \mathbin{\oplus} \mathit{xs}_R) &= \mathit{comb}_1 \ ((\mathit{orig} \triangleaux) \ \mathit{xs}_L, (\mathit{orig} \triangleaux) \ \mathit{xs}_R) \\
(\mathit{orig} \triangleaux) \ (\mathit{change} \ (c, \mathit{xs})) &= \mathit{comb}_2 \ (c, (\mathit{orig} \triangleaux) \ \mathit{xs})
\end{align*}
\]

Through the construction in Example 6.2, these two formulas can be unified into a single lifting problem with the following \( op \), where the complementary input is either an element in the change set of \( \mathit{change} \) or a token "d&c" never used before.

\[
\begin{align*}
op \ (c, (\mathit{xs}_L, \mathit{xs}_R)) &\triangleq \textbf{if } c = "d&c" \textbf{then } \mathit{xs}_L \mathbin{\oplus} \mathit{xs}_R \textbf{ else } \mathit{change} \ (c, \mathit{xs}_L)
\end{align*}
\]

Let \( n \) be the length of the initial list. A segment tree invokes \( \mathit{comb}_1 \) and \( \mathit{comb}_2 \) \( O(n \log n) \) times when processing each operation (either a query or a change). Therefore, the resulting program must be \( O(n \log n) \) time per operation under the efficiency condition. More details on this guarantee can be found in Appendix B.
we add operators for accessing and constructing tuples (i.e., \((1:30\) Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu

Analyzing the efficiency of the resulting program (Section 6.1, assuming the efficiency condition). 

\(L\) generate unrealizable subtasks (Section 5, assuming the compressing property of \(L_{aux}\)).

Scalar calculations by using the branch operator in a nested manner. Similar to the case of \(L\) operators (e.g., \(\neg\)), includes basic arithmetic operators (e.g., \(+\) and \(-\)).

Domain-specific languages. A lifting problem requires two languages \(L_{aux}\) and \(L_{comb}\) to specify the spaces of candidate auxiliary programs and combinators, respectively (Definition 3.1).

Language \(L_{aux}\) in our implementation (Figure 17) is the language of DeepCoder [Balog et al. 2017]. It includes 17 list-related operators, including common higher-order functions (e.g., map and filter) and several operators that perform branching and looping internally (e.g., count and sort). Because the language of DeepCoder does not support producing tuples, we add an operator for constructing tuples at the top level to cope with the cases where multiple auxiliary values are required.

**Example 7.1.** Operator \(\text{scanl}\) in our \(L_{aux}\) receives a binary operator \(\oplus\) and a list. It constructs all non-empty prefixes of the list and reduces each of them to an integer via \(\oplus\), as shown below.

\[
\text{scanl} (\oplus) [x_{s_1}, \ldots, x_{s_n}] \triangleq [x_{s_1}, x_{s_1} \oplus x_{s_2}, \ldots, x_{s_1} \oplus \cdots \oplus x_{s_n}]
\]

Using this operator, the maximum prefix sum of list \(xs\) can be implemented as \(\text{max} (\text{scanl} (\oplus) xs)\).

Language \(L_{comb}\) in our implementation (Figure 18) is the language of the conditional arithmetic domain in SyGuS-Comp [Alur et al. 2019], a world-wide competition for program synthesis. It includes basic arithmetic operators (e.g., + and \(\times\)), comparison operators (e.g., = and \(\leq\)), Boolean operators (e.g., \(\neg\) and \(\wedge\)), and the branch operator if-then-else. This language can express complex scalar calculations by using the branch operator in a nested manner. Similar to the case of \(L_{aux}\), we add operators for accessing and constructing tuples (i.e., \((S, S)\) and \(N_{T.i}\)) to deal with the cases requiring multiple auxiliary values.

These languages match the assumption we used when analyzing the probability for AutoLifter to generate unrealizable subtasks (Section 5, assuming the compressing property of \(L_{aux}\)) and when analyzing the efficiency of the resulting program (Section 6.1, assuming the efficiency condition).

---

• For the compressing property, programs in $\mathcal{L}_{aux}$ all map an integer list (that can be arbitrarily large) to a constant-sized integer tuple. Therefore, their input domains are far larger than their output domains when large enough lists are considered and every integer is bounded within a fixed range\(^5\).

• For the efficiency condition, one can verify that every program in $\mathcal{L}_{aux}$ runs in constant time on a constant-sized list, and every program in $\mathcal{L}_{comb}$ runs in constant time. Therefore, the efficiency condition is satisfied, and our implementation can provide all efficiency guarantees established in Section 6.

**Verification.** Since AutoLifter applies the CEGIS framework to solve leaf subtasks (Section 4.3), it requires corresponding verifiers to generate counter-examples for incorrect programs. We implement these verifiers using bounded model checking [Biere et al. 2003]. Specifically, we assume the original program is implemented in C++. Given a candidate program written in our domain-specific languages (Figures 18 and 17), our verifier will first translate it and its specification into C++ and then apply CBMC [Kroening et al. 2023], a popular bounded model checker, to verify whether the specification is satisfied. In this procedure, we consider only lists within a length limit (6 by default) to bound the depth of loops and recursions.

Besides, since it is time-consuming to perform bounded-model checking, in each CEGIS iteration, we will first test the candidate program using a set of random examples and will invoke the above verifier only when the candidate program satisfies all of these examples. Note that the random testing itself can provide a probabilistic correctness guarantee when the number of tested examples is large enough. Details on this point can be found in Appendix A.3.

Our implementation generates random examples as follows.

• Every tuple is generated by recursively generating its components.
• Every list is generated by (1) uniformly drawing its length from integers in $[0, 10]$, and then (2) recursively generating every element.
• Every integer is uniformly drawn from integers in $[-5, 5]$ by default. This range may change for tasks with specialized requirements. For example, some tasks in the dataset collected by Farzan and Nicolet [2021b] consider only 01 lists. Correspondingly, the range of integers is set to $[0, 1]$ on these tasks.

**Other configurations.** We implement the decomposition system in AutoLifter using the greedy strategy (Algorithm 1). In each decomposition, our implementation considers only the first solution synthesized from the first subtask and will fail once an unrealizable subtask is generated.

A synthesizer based on input-output examples is required to solve the leaf subtasks for comb (Algorithm 3, Section 4.3). Our implementation uses PolyGen [Ji et al. 2021], a state-of-the-art synthesizer on the conditional arithmetic domain.

The example-based synthesizer for aux (Algorithm 3, Section 4.3) is configured by an integer $lim_c$, representing the maximum number of components in the top-level combination. We set $lim_c$ to 4 by default because 4 auxiliary values are already enough for most known lifting problems.

8 EVALUATION

To evaluate AutoLifter, we report two experiments to answer the following research questions.

• **RQ1**: How effective does AutoLifter solve lifting problems?
• **RQ2**: Does AutoLifter outperform existing synthesizers in applying D&C?
• **RQ3**: Does AutoLifter outperform existing synthesizers in applying single-pass?

\(^5\)The actual case is relatively more complex because many programs in $\mathcal{L}_{aux}$ may enlarge the range of integers. We shall discuss this point in Section 9.
- **RQ4**: How does observational covering affect the performance of *AutoLifter*?

### 8.1 Experimental Setup

**Baseline Solvers.** In our evaluation, we compare *AutoLifter* with two general-purpose synthesizers, *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018]. Both of them can be applied to solve lifting problems and can be instantiated as synthesizers for D&C-like paradigms as *AutoLifter* does.

- **Enum** [Alur et al. 2013] is an enumerative solver. Given a lifting problem, *Enum* enumerates all possible \((aux, comb)\) from small to large until a valid one is found.
- **Relish** [Wang et al. 2018] is a state-of-the-art synthesizer for relational specifications. It first excludes many invalid programs via a data structure namely hierarchical finite tree automata and then searches for a valid program among the automata.

In our evaluation, we re-implement both *Enum* and *Relish* because of the reasons below.

- The original implementation of *Enum* cannot solve lifting problems in our dataset. Specifically, this implementation requires the synthesis task to be specified in the SyGuS input format [Padhi et al. 2021]. However, this format does not support list-related operators and thus cannot express the language \(L_{comb}\) we use (Figure 18).
- The input format of the original implementation of *Relish* does not support the languages we use, and it is difficult to update this implementation to match our demand.

We re-implement these two synthesizers using the same setting as our implementation of *AutoLifter* (Section 7). Specifically, we use the same domain-specific languages to specify the program spaces and use the same verifier to generate counter-examples for incorrect candidate programs.

Besides, we also compare *AutoLifter* with two state-of-the-art specialized synthesizers.

- **Parsynt** [Farzan and Nicolet 2017, 2021b] is a specialized synthesizer for D&C that relies on syntax-based program transformations. Specifically, it requires the original program to be single-pass, then extracts \(aux\) by transforming the loop body using pre-defined transformations, and at last, synthesizes a corresponding \(comb\) via an existing synthesizer. There are two versions of *Parsynt* available, denoted as *Parsynt17* [Farzan and Nicolet 2017] and *Parsynt21* [Farzan and Nicolet 2021b], where different syntax-based program transformations are used. We consider both versions of *Parsynt* in our evaluation.
- **DPASyn** [Pu et al. 2011] is a specialized synthesizer for single-pass. It reduces the application task of single-pass to a Sketch problem [Solar-Lezama 2013], solves this task using existing Sketch solvers, and also includes specialized optimizations for dynamic programming programs. We use a re-implementation based on *Grisette* [Lu and Bodík 2023] (provided by the authors of *DPASyn*) in our evaluation.

We list two worth-noting details on these two specialized base solvers as follows.

- *Parsynt* and *DPASyn* cannot be applied to each other’s tasks because they both rely on the domain-specific property of their respective paradigms. Specifically, the program transformations used by *Parsynt* rely on the specific relationship between D&C and single-pass, and the reduction to the Sketch problem in *DPASyn* is specifically designed for single-pass.
- These two solvers provide the same efficiency guarantee on the resulting program as *AutoLifter*. Specifically, both *AutoLifter* and *Parsynt* ensure that the resulting D&C program runs in \(\Theta(n/p)\) time on a list of length \(n\) and \(p \leq n/\log n\) processors; both *AutoLifter* and *DPASyn* ensure that the resulting single-pass program runs in \(\Theta(n)\) time on a list of length \(n\).

**Dataset.** Our evaluation is conducted on a dataset of 96 tasks of applying D&C-like algorithmic paradigms (Table 4). These tasks are related to the four algorithmic problems.
Table 4. The profile of synthesis tasks considered in our evaluation.

<table>
<thead>
<tr>
<th>Problem</th>
<th>D&amp;C</th>
<th>Single-pass</th>
<th>Longest Segment</th>
<th>Segment Tree</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Task</td>
<td>36</td>
<td>39</td>
<td>8</td>
<td>13</td>
<td>96</td>
</tr>
</tbody>
</table>

- **Problem 1: applying D&C to a program.** We collect 36 such tasks from the datasets of previous studies [Bird 1989a; Farzan and Nicolet 2017, 2021b]6, including all tasks used by Farzan and Nicolet [2017] and Bird [1989a] and 12 out of 22 tasks used by Farzan and Nicolet [2021b]. The other 10 tasks used by Farzan and Nicolet [2021b] are out of the scope of AutoLifter because they cannot be reduced to lifting problems. They require a more general form of D&C where the divide operator is not determined, making our reduction inapplicable.

- **Problem 2: applying single-pass to a program.** We consider the tasks used in the evaluation of DPASyn [Pu et al. 2011] and include 4 out of 5 tasks into our dataset. The last task includes multiple input lists and is not supported by our current implementation. Besides, we construct a series of tasks from our D&C tasks as a supplement. For each D&C task, a single-pass task with the same original program is constructed7. These tasks correspond to a possible application for removing the restriction on the input program from existing transformation-based synthesizers for D&C: one can first apply AutoLifter to get a single-pass program and then use existing synthesizers to generate a D&C program.

- **Problem 3: Longest Segment Problem.** Zantema [1992] proposes three algorithmic paradigms for longest segment problems and discusses 3, 1, and 4 example tasks, respectively. For each example task, we include the task of applying the respective paradigm in our dataset.

- **Problem 4: applying segment trees to answer queries on a specific property of a segment in a list.** We collect some such tasks online because no previous work on segment trees provides a dataset. Specifically, we collect 13 tasks by searching on Codeforces (https://codeforces.com/), a website for competitive programming, using keywords "segment tree" and "lazy propagation"8, and include all these tasks in our dataset.

In our evaluation, we assume the correct paradigm for each task is available and thus evaluate AutoLifter (as well as baseline solvers Enum and Relish) by directly applying the corresponding instantiation to each task. This setting is the same as the previous studies on automatically applying algorithmic paradigms [Acar et al. 2005; Farzan and Nicolet 2021b; Lin et al. 2019; Morita et al. 2007; Raychev et al. 2015] and corresponds to the usage scenario that the users need to decide which paradigm to use by themselves. We shall discuss the selection of paradigms in Section 9.

**Configuration.** Our experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor. Every execution is under a time limit of 300 seconds and a memory limit of 8 GB.

### 8.2 RQ1: Comparison of Synthesizers for Lifting Problems

**Procedure.** In this experiment, we compare AutoLifter with Enum and Relish, the two baseline solvers that support solving general lifting problems, on all tasks in our dataset. We consider three different aspects when comparing these solvers.

- The effectiveness of synthesis, measured by the number of solved tasks.
- The efficiency of synthesis, measured by the average time cost of successful synthesis.

---

6The original dataset of Parsynt21 contains two bugs in task longest_1(0*)2 and longest_odd_(0+1) that were introduced while manually rewriting the original program into single-pass. These bugs were confirmed by the original authors, and we fixed them in our evaluation. This also demonstrates that writing a single-pass program is difficult and error-prone.

7A duplicated task involved by both DPASyn and our D&C dataset is ignored in this construction.

8A common alias of segment trees.
Table 5. The results of comparing *AutoLifter* with *Enum* and *Relish*.

| Solver | D&C | | | | Single-pass | | |
|---|---|---|---|---|---|---|
| | #Solved | Time base | Time out | Time res | #Solved | Time base | Time out | Time res |
| AutoLifter | 29/36 | 20.01 | ×1.000 | | | 33/39 | 8.61 | ×1.000 |
| Enum | 5/36 | 46.76 | 0.247 | ×0.981 | 9/39 | 9.544 | 0.337 | ×1.090 |
| Relish | 12/36 | 34.75 | 7.283 | ×0.986 | 16/39 | 18.03 | 4.366 | ×1.018 |

| Solver | Longest Segment | | | | Segment Tree | | |
|---|---|---|---|---|---|---|
| | #Solved | Time base | Time out | Time res | #Solved | Time base | Time out | Time res |
| AutoLifter | 7/8 | 22.87 | ×1.000 | 13/13 | 43.30 | ×1.000 |
| Enum | 1/8 | 14.99 | 0.530 | ×0.981 | 4/13 | 115.1 | 19.84 | ×1.001 |
| Relish | 3/8 | 4.229 | 2.377 | ×0.980 | 7/13 | 86.47 | 42.20 | ×1.002 |

| Solver | Total | | | | | | |
|---|---|---|---|---|
| | #Solved | Time base | Time out | Time res |
| AutoLifter | 82/96 | 20.17 | ×1.000 |
| Enum | 19/96 | 41.84 | 4.430 | ×1.035 |
| Relish | 38/96 | 34.98 | 14.07 | ×1.002 |

- The efficiency of the resulting program, measured by the time cost of the resulting program on a randomly generated test suite. Specifically, we calculate this time cost in three steps.
  1. For each paradigm considered in our dataset, we implement a template in C++ and an automatic translator to fill the synthesis result of lifting problems into this template.
  2. For each task in our dataset, we construct a test suite by randomly generating 5 different inputs. These inputs are generated in the same way as random examples (Section 7), except the length of lists is fixed to $10^7$. We believe such an input scale is large enough to reflect the efficiency difference between different programs.
  3. For each successful synthesis, we first complete the synthesis result into an executable program using the translator of the corresponding paradigm, then execute this program on every input in the test suite of the corresponding task, and at last record the average time cost on these inputs (with the IO cost excluded).

**Result.** The results of this experiment are summarized in Table 5, organized as follows.

- Column #Solved reports the number of tasks solved by each solver.
- Columns Time base and Time out report the average time costs of each baseline solver and *AutoLifter*, respectively. Only those tasks solved by both the baseline solver and *AutoLifter* will be considered when calculating these time costs.
- Column Time res reports the relative time cost of the resulting programs of each baseline solver compared with the resulting programs of *AutoLifter*. Specifically, for each task solved by both the baseline solver and *AutoLifter*, we calculate the relative time cost as ratio $t_{\text{base}}/t_{\text{ours}}$, where $t_{\text{base}}$ and $t_{\text{ours}}$ denote the time costs of the resulting programs of the baseline solver and *AutoLifter*, respectively. Then, we report the geometric average of these ratios in Column Time res. Here, an average ratio larger than 1 means that the resulting program of *AutoLifter* is more efficient than that of the baseline solver.

Besides, we also manually analyze the synthesis results from two aspects, as shown below.
• Since the verifier in our implementation (which is a combination of bounded model checking and random testing, Section 7) does not ensure the full correctness, we manually verify all synthesis results and confirm that they are all completely correct.

• We manually verify the applications of our decomposition methods in every execution of AutoLifter and confirm that no unrealizable subtask is generated from realizable lifting problems, which matches our probabilistic completeness guarantee (Theorem 5.6).

The results in Table 5 demonstrate that AutoLifter significantly outperforms the baseline solvers. In the sense of solving lifting problems, AutoLifter not only solves many more tasks but also uses a much smaller time cost; and in the sense of synthesizing efficient programs, there is no significant efficiency difference between the resulting programs of AutoLifter and those of the baseline solvers, where the relative difference never exceeds 2%.

It is expected that there is no significant efficiency gap between the resulting programs. Specifically, the efficiency of a program is roughly determined by two factors, its time complexity and the constant factor in its time cost. Both of these factors must be the same (or close) when comparing the resulting programs of AutoLifter, Enum, and Relish.

• The time complexity of these resulting programs must be the same because our efficiency condition (Section 6.1) ensures that, for any paradigm considered in our evaluation, the time complexity of the resulting program must be the same no matter which aux and comb are synthesized from the domain-specific languages we use (Figures 18 and 17).

• The constant factor of these resulting programs must be close because (1) the constant factor is majorly determined by the time cost of the synthesized comb, (2) the time cost of comb is closely related to its size, and (3) solvers AutoLifter, Enum, and Relish all ensure to synthesize a comb whose size is close to the smallest. Specifically, both Enum and Relish ensure that the total size of the synthesized aux and comb is minimized, and PolyGen (the client synthesizer in AutoLifter for synthesizing comb) ensures that the size of the synthesized program is close to the smallest under the theory of Occam learning [Blumer et al. 1987].

AutoLifter fails on 14 out of 96 tasks in our dataset, all of which are unrealizable because of the limited expressiveness of the default languages used in our implementation. These tasks require specialized operators such as regex matching on an integer list and the power operator on integers. These operators are not included in the general-purpose languages we used since they are not common in the domains of lists and integer arithmetic.

After supplying missing operators, AutoLifter can solve 13 more tasks and find a valid auxiliary program for the last remaining task. The last failed task is longest_odd_(0+1) constructed by Farzan and Nicolet [2021b], on which AutoLifter fails because PolyGen times out in finding a corresponding combinator. This result suggests that AutoLifter can be further improved if missing operators can be inferred automatically, for example, by incorporating those transformation-based approaches and extracting useful operators from the source code. This is a direction for future work.

8.3 RQ2: Comparison with Synthesizers for Divide-and-Conquer Procedure

In this experiment, we compare AutoLifter with Parsynt, the baseline solver specialized for D&C, on all D&C tasks in our dataset. The details of the experiment setup are shown below.

• Since Parsynt requires the original program to be single-pass (Section 8.1), we provide a single-pass implementation of the original program for each task to invoke Parsynt10. Note

---

9A gold medal winner in international programming competitions helped us to verify the synthesized programs.

10For those tasks taken from Parsynt, we use the program in its original evaluation and fix the two bugs we found.
that this setting favors Parsynt because (1) many programs cannot be implemented as single-pass unless some auxiliary values are manually introduced (Section 2.3), and (2) Parsynt can access those auxiliary values provided in the single-pass implementation.

- Since there are two versions of Parsynt available (i.e., Parsynt17 and Parsynt21, mentioned in Section 8.1), we consider both of them in this experiment.
- We failed in installing Parsynt17 because of some dependency issue. This issue has been confirmed by the authors of Parsynt17 but has not been solved yet. Therefore, we compare with Parsynt only on its original dataset (which is a subset of ours) using the evaluation results reported by its original paper [Farzan and Nicolet 2017].
- Similar to the first experiment (Section 8.2), we consider three metrics when comparing AutoLifter with Parsynt, including the number of solved tasks, the time cost of successful synthesis, and the time cost of the resulting programs. When evaluating the time cost of the resulting programs, we will fill the synthesis result of Parsynt into our template of D&C because the original implementation of Parsynt does not provide a default template.

Results. The results of this experiment are summarized in Table 6, organized as follows.

- Columns #SolvedBase and #SolvedOurs report the number of tasks solved by each version of Parsynt and AutoLifter, respectively.
- Columns #TimeBase and #TimeOurs report the average time costs of Parsynt and AutoLifter, respectively, and Column #TimeRes report the relative time cost of the resulting programs of Parsynt compared with the resulting programs of AutoLifter. Values in these columns are calculated in the same way as the corresponding columns in Table 5.
- Column #AuxSP report the ratio of the number of auxiliary values provided in the input single-pass program to the number of auxiliary values used in the D&C program synthesized by Parsynt. A larger value here means that Parsynt requires more extra inputs.

Besides, the value of cell (Parsynt17, TimeRes) in this table is unavailable because the original paper of Parsynt17 does not provide the full synthesis results.

The results in Table 6 show that compared with Parsynt, AutoLifter can offer competitive performance on synthesizing D&C programs while using significantly less information from the input. Specifically, when compared with Parsynt17, AutoLifter solves the same number of tasks with a smaller time cost; and when compared with Parsynt21, AutoLifter solves more tasks and synthesizes more efficient D&C programs, though requiring slightly more time for synthesis. Please note that in this comparison, Parsynt takes much more input than AutoLifter, including 40.54%-58.62% of those necessary auxiliary values.

Now, we would like to discuss more the efficiency of the resulting programs. Table 6 shows that, although AutoLifter and Parsynt21 provide the same guarantee on the time complexity of the resulting programs (Section 8.1), the resulting programs of AutoLifter tend to be more efficient than those of Parsynt21, with a relative advantage of about 20%. One important reason for this result is that the resulting programs of Parsynt21 tend to use more auxiliary values than those of AutoLifter, leading to an extra time cost for calculating auxiliary values. In this experiment, the...
resulting programs of Parsyn21 never use fewer auxiliary values than those of AutoLifter and use more auxiliary values on 10 tasks.

Parsynt21 tends to use more auxiliary values because its syntax-based program transformations may be misled by the source code of the original program. For example, the original program of task line_sight (abbreviated as ls) checks whether the last element is the maximum of the list. It can be implemented as single-pass with an auxiliary program max returning the maximum of a list, because \( \text{ls}(xs + [v]) = v \geq (\text{max} \, xs) \). Given this program, Parsynt will extract the last element of a list as an auxiliary value because the last visited element \( v \) is used in the loop body. However, this value is not necessary because \( \text{ls}(l_1 + l_2) \) is always equal to \( (\text{ls} \, l_2) \land (\text{max} \, l_1 \leq \text{max} \, l_2) \). AutoLifter can generate this simpler solution because it finds auxiliary values by enumerating programs in \( \mathcal{L}_{\text{aux}} \) instead of transforming the source code of the original program.

At last, we find that when applying D&C, the issue of missing operators on AutoLifter (Section 8.2) can be alleviated by combining AutoLifter with Parsynt21. Although the default languages we use are not expressive enough for 7 D&C tasks in our dataset, our languages are enough for applying single-pass to the original programs of 5 tasks among these tasks. AutoLifter can successfully synthesize single-pass programs for these 5 tasks, and then Parsynt21 can synthesize D&C programs for 4 among them. In this way, the combination of AutoLifter and Parsynt21 can solve 33 out of 36 tasks, outperforming both individual solvers.

8.4 RQ3: Comparison with Synthesizers for Single-Pass

Procedure. In this experiment, we compare AutoLifter with DPASyn, the baseline solver specialized for single-pass, on all single-pass tasks in our dataset. Similar to the previous experiments, we consider three metrics in this experiment, including the number of solved tasks, the time cost of successful synthesis, and the time cost of the resulting programs. When evaluating the time cost of the resulting programs, we will fill the synthesis result of DPASyn into our template of single-pass because DPASyn does not provide a default template.

We consider two different configurations of DPASyn in this comparison, a normal configuration (denoted as DPASyn\(_n\)) for establishing a fair comparison and an enhanced configuration (denoted as DPASyn\(_e\)) for better revealing the performance of DPASyn.

- DPASyn\(_n\) uses the same language as AutoLifter to specify the program space.
- DPASyn\(_e\) uses a more compact program space. We consider this configuration because, as a Sketch-based synthesizer, the time cost of DPASyn will increase dramatically when the scale of the target program increases. However, the language \( \mathcal{L}_{\text{comb}} \) we use (Figure 18) is so basic that a large program may be necessary for even simple tasks. Therefore, to better evaluate DPASyn, we customize the program space of each task for DPASyn\(_e\) by (1) adding operators max and min and (2) excluding those operators that are not used.

Note that the comparison between AutoLifter and DPASyn\(_e\) favors the latter because DPASyn\(_e\) needs only to explore a much smaller program space.

Results. The results of this experiment are summarized in the right-side table, organized in the same way as the table of the first experiment (Table 5, Section 8.2). These results show that AutoLifter outperforms both versions of DPASyn. Specifically, AutoLifter solves more tasks with a smaller time cost, and there is no significant efficiency difference between the resulting programs of AutoLifter and DPASyn.
Table 7. The results of comparing AutoLifter with AutoLifterOE.

<table>
<thead>
<tr>
<th>Solver</th>
<th>D&amp;C</th>
<th></th>
<th>Single-pass</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Solved</td>
<td>TimeBase</td>
<td>TimeOurs</td>
<td>TimeRes</td>
<td>#Solved</td>
<td>TimeBase</td>
<td>TimeOurs</td>
<td>TimeRes</td>
<td>#Solved</td>
<td>TimeBase</td>
<td>TimeOurs</td>
<td>TimeRes</td>
</tr>
<tr>
<td>AutoLifter</td>
<td>29/36</td>
<td>30.79</td>
<td>1.000</td>
<td>33/39</td>
<td>8.861</td>
<td>1.000</td>
<td>1.010</td>
<td>0.985</td>
<td>33/39</td>
<td>8.861</td>
<td>1.000</td>
<td>1.010</td>
</tr>
<tr>
<td>AutoLifterOE</td>
<td>13/36</td>
<td>3.329</td>
<td>1.000</td>
<td>28/39</td>
<td>5.563</td>
<td>4.141</td>
<td>1.000</td>
<td>0.954</td>
<td>28/39</td>
<td>5.563</td>
<td>4.141</td>
<td>1.000</td>
</tr>
</tbody>
</table>

The advantage of AutoLifter majorly comes from its decomposition system, which decomposes the original task into subtasks on sub-programs with much smaller scales. In contrast, DPASyn directly searches for the whole target program, leading to a combinatorially larger search space.

8.5 RQ4: Comparison with the Variant without Observational Covering

Procedure. AutoLifter uses a specialized optimization named observational covering when synthesizing aux (Section 4.3). In this experiment, we conduct an ablation study to test the effect of this optimization. Specifically, we consider a variant of AutoLifter (denoted as AutoLifterOE) where the leaf subtasks of aux are solved by the pure OE without using observational covering. Then, we compare this variant with the default AutoLifter on all tasks in our dataset. Similar to the previous experiments, we consider three metrics in this comparison, including the number of solved tasks, the time cost of successful synthesis, and the time cost of the resulting programs.

Results. The results of this experiment are summarized in Table 7, organized in the same way as the table of the first experiment (Table 5, Section 8.2). These results demonstrate that observational covering significantly improves the efficiency of AutoLifter.

Note that even when observational covering is removed, AutoLifterOE still outperforms Enum and Relish (Table 5) and outperforms DPASyn on synthesizing single-pass programs (Table 7). This result also demonstrates the effectiveness of our decomposition system.

8.6 Qualitative Analysis of Selected Tasks

To further illustrate the effectiveness of AutoLifter, we shall discuss two tasks in our dataset and show that (1) AutoLifter can solve tasks that are difficult for previous transformation-based approaches, and (2) AutoLifter can solve algorithmic tasks that are different for human programmers.

Maximum segment product. The first task is named as maximum segment product (msp) [Bird 1989a], which is an advanced version of mss (Section 4.1). Given list \( x_s[1 \ldots n] \), the problem is to select a segment \( s \) from \( xs \) and maximize the product of values in \( s \).

It is not easy to calculate the maximum segment product by D&C. According to the experience in solving the mss task, one may choose the maximum prefix/suffix product as the auxiliary values. However, these two values are not enough. The counter-intuitive point here is that the maximum segment product is also related to the minimum prefix/suffix product. This is because both the minimum suffix product of the left half and the minimum prefix product of the right half can be
negative integers with large absolute values. Their product will flip back the sign, resulting in a large positive number. For example, the segment with the maximum product of $[-1, -5] add [-3, 0]$ is $[-5, -3]$, formed by the suffix with the minimum product of the left half (i.e., $[-5]$) and the prefix with the minimum product of the right half (i.e., $[-3]$).

*Parsynt* fails to solve this task as its transformation rules are not enough to extract these auxiliary values (related to the minimum) from the original program (related to the maximum). In contrast, *AutoLifter* successfully solves this task in 287.9 seconds (where 113.9 seconds are used by bounded model checking) and finds an auxiliary program as follows.

$$\text{aux } xs \triangleq (\text{max (scanl } \times \text{) } xs), \text{max (scanr } \times \text{) } xs), \text{min (scanl } \times \text{) } xs), \text{min (scanr } \times \text{) } xs), \text{head (scanr } \times \text{) } xs)$$

This program calculates five auxiliary values, corresponding to the maximum prefix product, the maximum suffix product, the minimum prefix product, the minimum suffix product, and the product of all elements, respectively. We omit the combinator synthesized by *AutoLifter* because it is large in scale but is straightforward from the synthesized auxiliary program.

**Longest segment problem 22-2.** The second problem is proposed by Zantema [1992], which is used as the second example on Page 22 of that paper. This problem is to find a linear-time program for the length of the longest segment $s$ satisfying $\text{min } s + \text{max } s > \text{length } s$ for a given list.

This problem is difficult even for professional programmers in competitive programming. It was set as a problem in 2020-2021 Winter Petrozavodsk Camp, a worldwide training camp representing the highest level of competitive programming. Only 26 out of 243 participating teams solved this problem within the 5-hour competition.

The third algorithmic paradigm proposed by Zantema [1992] can be applied to solve this problem. The synthesis task is to find an auxiliary program aux and a combinator comb such that the formula below is satisfied for any lists $x_{SL}, x_{SR}$ and integer $v$ satisfying $v < \text{min } x_{SL} \land v \leq \text{min } x_{SR}$.

$$(\text{orig } \triangleq \text{aux}) \ (x_{SL} \radd [v] \radd x_{SR}) = \text{comb} \left( v, \left( (\text{orig } \triangleq \text{aux}) \ x_{SL}, (\text{orig } \triangleq \text{aux}) \ x_{SR} \right) \right)$$

where *orig* is an arbitrary reference program returning the length of the longest valid segment. However, it is difficult to find proper aux and comb satisfying the above formula. We encourage the readers to try to solve this task before moving to the discussion below.

*AutoLifter* can find an auxiliary program aux $xs \triangleq (\text{length } xs, \text{max } xs)$ and a correct combinator comb in 100.0 seconds. The synthesized comb includes 152 AST nodes and is formed by several components dealing with different cases. Here, we only explain the component for calculating the expected output under the condition that $\text{max } x_{SL} \geq \text{max } x_{SR}$, as shown below.

$$\text{comb} \left( v, (\text{res}_{L}, \text{res}_{R}) \right) \triangleq \begin{cases} \text{max(}l_{SP_{R}}, \text{min(}l_{SP_{L}} + l_{EN_{R}} + 1, v + \text{max}_{L} - 1)) & \text{if } v + \text{max } x_{SL} > \text{length } x_{SL} + 1 \\ \text{max(}l_{SP_{L}}, l_{SP_{R}}) & \text{otherwise} \end{cases}$$

where $\text{res}_{L}$ is unfolded to $(l_{SP_{L}}, (l_{EN_{L}}, \text{max}_{L}))$, $\text{res}_{R}$ is unfolded to $(l_{SP_{R}}, (l_{EN_{R}}, \text{max}_{R}))$

assuming $\text{max } x_{SL} \geq \text{max } x_{SR}$

**Case 1:** $(\text{max } x_{SL} \geq \text{max } x_{SR}) \land (v + \text{max } x_{SL} > \text{length } x_{SL} + 1)$. There are only three possible cases for the longest valid segment: the longest valid segment $s_{L}$ in $x_{SL}$, the longest valid segment $s_{R}$ in $x_{SR}$, or the longest valid segment $s_{0}$ including element $v$. In this case, $s_{L}$ is no longer than $s_{0}$ because segment $x_{SL} \radd [v]$ is valid under the condition that $v + \text{max } x_{SL} > \text{length } x_{SL} + 1$. Therefore, the longest valid segment of the whole list must be the longer one between $s_{R}$ and $s_{0}$. Since the length of $s_{R}$ is known as $l_{SP_{R}}$, the remaining task is to get the length of $s_{0}$.

An observation is that segment $x_{SL} \radd [v]$ already achieves the maximum possible $\text{min } s + \text{max } s$ among segments including $v$ because (1) $\text{min } s$ must be $v$, the minimum of the whole list, and (2)
max s must be no larger than max xs_L, the maximum of the whole list under the condition that max xs_L ≥ max xs_R. Therefore, s_o is the longest segment expanded from xs_L + [v] until the length limit (i.e., v + max xs_L) is reached or the whole list is used up, that means, the length of s_o must be min(length s + length s + 1, v + max s - 1).

Case 2: (max xs_L ≥ max xs_R) ∧ (v + max xs_L ≤ length xs_L + 1). In this case, s_o is no longer than s_L, so the longest valid segment of the whole list is the longer one between s_L and s_R, and the result is max(length s_L, length s_R). This property can be proved in two steps. First, s_o must be no longer than xs_L, as shown by the derivation below.

\[
\text{length } s_o \leq v + \text{max } s_o - 1 \leq v + \text{max } x_{s_L} - 1 \leq \text{length } x_{s_L}
\]

The first inequality uses the fact that v is the minimum of the whole list, the second inequality uses the fact that max xs is the maximum of the whole list under the condition that max xs_L ≥ max xs_R, and the third inequality uses the condition that v + max xs_L ≤ length xs_L + 1.

Second, since s_o is no longer than xs_L, there exists another segment s’ that includes the maximum of xs_L and has the same length as s_o. As shown by the derivation below, s’ must be valid as well. Therefore, s’ is no longer than s_L, implying that s_o is no longer than s_L.

\[
\text{min } s' + \text{max } s' \geq v + \text{max } s_o > \text{length } s_o = \text{length } s'
\]

As we can see, the correct comb here relies on several tricky properties. Finding this program is challenging for a human user. In comparison, AutoLifter can solve this problem quickly.

9 DISCUSSION

Selecting algorithmic paradigms. As discussed in Section 6, AutoLifter can be instantiated as a series of synthesizers, each for applying a specific D&C-like paradigm. However, the presence of multiple instantiations brings another problem in usage, that is, how to select a proper instantiation for a practical task. Currently, this selection problem is not yet an issue because the number of available instantiations of AutoLifter is not large (6 in our implementation): we can simply try all available instantiations in order or in parallel until the task of interest is solved. In the future, when more instantiations are developed, it may be necessary to design an automated approach to select among all possible choices. This is a direction for future work.

Besides, we believe AutoLifter can still have a significant practical effect even if the selection problem is left to the user.

- On the one hand, some paradigms are so important that even a specialized synthesizer is still valuable. For example, the problem of automatic parallelization has long been studied in literature [Farzan and Nicolet 2017; Fedyukovich et al. 2017; Morita et al. 2007; Raychev et al. 2015]. The instantiation of AutoLifter on the D&C paradigm solves this problem, and compared with previous approaches, this instantiation of AutoLifter is the first one that does not require the original program to be single-pass.

- On the other hand, it is usually not difficult for the user to select among paradigms because the application scope of different paradigms is usually clear. For example, the D&C paradigm is usually used for parallelization, where the goal is to achieve a sublinear time complexity on multiple processors; the incrementalization paradigm is only available for incremental tasks where the original program will be executed multiple times on a series of similar inputs.

Verification. In this paper, we focus on designing an effective synthesizer for applying algorithmic paradigms and do not consider designing specialized verifiers. Instead, our approach can be combined with any off-the-shelf verifier to provide a correctness guarantee on the synthesis result.
Although our current implementation (Section 7, where bounded model checking is used) cannot ensure full correctness, it works well in our evaluation. As discussed in Section 8.2, we manually verify all synthesis results and confirm that all of them are completely correct. This result shows that our current implementation can already provide reliable synthesis results in practice.

Besides, even when the user decides to manually verify the synthesized program, we believe this task will still be easier than solving the algorithmic task by the users themselves. On the one hand, an algorithmic task can be extremely difficult (e.g., the second task discussed in Section 8.6), and AutoLifter can provide a candidate solution for the user to check, which is correct with a high probability. On the other hand, AutoLifter ensures to synthesize simple programs (Theorem 5.8 and the property of PolyGen [Ji et al. 2021]), making its result usually easy to understand.

In the future, we believe the ability of verifiers will be continuously improved and will be able to verify more and more complex programs. At that time, by combining with those more advanced verifiers, AutoLifter can potentially provide a stronger correctness guarantee on its result.

The greedy strategy v.s. the backtracking strategy. The effectiveness of AutoLifter comes from its decomposition system, which uses approximate specifications to break the dependency among sub-programs. There are two possible strategies for implementing such a decomposition system (Section 5). The greedy strategy used in our implementation considers only the first solution of each subtask and will fail once an unrealizable subtask is generated. In comparison, the backtracking strategy will roll back and try to find other solutions each time an unrealizable subtask is met. It will not fail but will be inefficient if too many unrealizable subtasks are generated.

At first glance, the backtracking strategy may seem like a good choice because it can realize a complete synthesis, that is, the synthesis will never fail on a realizable lifting problem. However, it is not easy to implement this strategy because there are still two challenges remaining.

- First, to decide whether to roll back the decomposition, the backtracking strategy requires checking whether the current subtask is unrealizable. Although some research progress has been made [Hu et al. 2020; Kim et al. 2021], proving unrealizability for a program synthesis task is still time-consuming in practice.
- Second, some efficiency synthesis techniques will be unavailable for synthesizing aux if we want to achieve completeness using the backtracking strategy. Specifically, the synthesizer for aux needs to ensure that every possible auxiliary program will be considered after backtracking enough times, otherwise, some valid solutions may be missed. However, most synthesis techniques (e.g., observational equivalence and observational covering we used) fail to satisfy this requirement. They are designed only for synthesizing a single program and may skip many non-optimal programs during the synthesis for efficiency.

Because of the above challenges in implementing the backtracking strategy, we use the greedy strategy to implement AutoLifter. Although in theory, the greedy strategy may fail in solving realizable lifting problems, both our probabilistic analysis (Section 5) and our evaluation results (Section 8) suggest that this failure seldom happens in practice.

The compressing property. When analyzing the effectiveness of AutoLifter (Section 5), we utilize the compressing property of practical lifting problems, that is, the original program and auxiliary programs usually map from a large input domain to a small output domain. Our analysis does not put a strict restriction on how much the input domain should be larger than the output domain, instead, it shows how the effectiveness of AutoLifter is gradually affected by a larger input domain. Specifically, when the input domain becomes larger (compared to the output domain), the mismatch factor of a lifting problem will become larger, making the unrealizable rate of AutoLifter become smaller (Theorem 5.4). This unrealizable rate will finally converge to 0 when the size of the input domain approaches infinity (Theorem 5.6).
Table 8. The largest outputs from an integer list whose length $\leq n$ and element integers $\in [-m, m]$.

<table>
<thead>
<tr>
<th>Program</th>
<th>length</th>
<th>min</th>
<th>sndmin</th>
<th>sum</th>
<th>mps</th>
<th>mss</th>
<th>msp (Section 8.6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Output</td>
<td>$n$</td>
<td>$m$</td>
<td>$n \times m$</td>
<td>$n^m$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please note that the compressing property is never a sufficient condition for AutoLifter to succeed in synthesis. It is still possible to construct a realizable lifting problem where the input domain is much larger than the output domain but AutoLifter fails. We study the compressing property in this paper only to explain the effectiveness of AutoLifter and clarify the boundary of AutoLifter.

In the previous discussions, we simply regard those programs mapping from integer lists to a constant number of integers as compressing (Example 5.5 and Section 7). This claim holds only when the range of integers is assumed fixed and finite, but the practical situation is more complex. Many programs mapping from integer lists to integers will enlarge the range of integers, for example, the program calculating the sum of a list can return a number as large as $n \times m$ from a list whose length is no larger than $n$ and element integers are inside range $[-m, m]$. If such enlargement is not limited, non-compressing programs may exist, for example, there exist injective functions from integer lists to a single integer when the output integer is not bounded.

Luckily, the extreme case seldom happens in practice. Table 8 lists the output ranges of several programs mentioned before. As we can see, most of these programs enlarge the range of integers only polynomially and thus are still compressing because the number of integer lists is exponential to the list length and the input range. Even for the last program msp (maximum segment product, Section 8.6) that may return an exponentially larger integer, the compressing property still holds when the range of input integers is small. For example, the result of msp must be in the form of $2^a3^b5^c$ for $0 \leq a \leq 2n$ and $0 \leq b, c \leq n$ when only integers within $[-5, 5]$ are used in the input. Consequently, msp maps $11^n$ different input lists to only $O(n^3)$ different outputs, leading to the compressing property.

Besides the case of mapping data structures to scalar values, we observe that the compressing property also holds for many programs mapping between data structures. For example, the sorting program maps all permutations of length $n$ ($n!$ possibilities in total) to the same output $[1, 2, \ldots, n]$. This observation suggests interesting future work of applying AutoLifter to those tasks where not only auxiliary values but also auxiliary data structures are required.

10 RELATED WORK

Automatic applications of D&C-like paradigms. There have been previous studies on applying individual D&C-like algorithmic paradigms.

First, several approaches [Farzan and Nicolet 2017; Fedyukovich et al. 2017; Morita et al. 2007; Raychev et al. 2015] have been proposed to apply D&C. All of these approaches are based on syntax-based program transformations and require the input program to be implemented as single-pass. Compared with them, AutoLifter does not require single-pass implementations but can still offer competitive performance compared with the previous state-of-the-art (Section 8.3).

When applying D&C, we assume the list is always divided from the middle and thus focus on synthesizing the auxiliary program aux and the combinator comb. In this sense, Farzan and Nicolet [2021b] study the application of a more general form of D&C where the divide operator is to be synthesized as well. Their approach and ours are complementary because their approach requires a single-pass implementation of the original program while AutoLifter does not. A possible future direction is to combine these approaches with AutoLifter.
Second, Acar et al. [2005] proposes an approach for incrementalization. This approach records the execution trace of the original program as the auxiliary value and lets the combinator re-evaluate only those operations affected by the change. Consequently, this approach can generate an efficient program only when the execution trace of the original program is affected little by the change. However, it can be difficult to satisfy this requirement. For example, in the incrementalization task for \textit{sndmin} (Section 3.1), the resulting program generated from the natural implementation of \textit{sndmin} (Figure 1) will trace into the sorting function and thus runs in $O(\log n)$ time per change, much slower than the expected solution that runs in constant time per change (Figure 8).

Both \textit{AutoLifter} and Acar et al. [2005]’s approach have their advantages in automatic incrementalization. \textit{AutoLifter} does not rely on the source code of the original program and thus can generate efficient results regardless of the user-provided implementation. However, when a proper original program is given, Acar et al. [2005]’s approach can construct incremental programs for extremely difficult tasks such as generating dynamic data structures requiring hundreds of lines of code [Acar et al. 2009], where even the decomposed subtasks generated by \textit{AutoLifter} are still out of the scope of existing synthesizers. Scaling up inductive synthesis to these complex programs is future work.

Third, Pu et al. [2011] propose an approach named \textit{DPASyn} to apply single-pass. This approach reduces the synthesis task to a Sketch problem and solves it via existing Sketch solvers. Compared with this approach, \textit{AutoLifter} involves a decomposition system to decompose the synthesis task into subtasks with much smaller scales and thus greatly reduces the search space. Our evaluation results demonstrate the effectiveness of \textit{AutoLifter} (Section 8.4).

At last, there exist multiple approaches that do not support finding necessary auxiliary values when the paradigm cannot be directly applied. The related paradigms include D&C [Ahmad and Cheung 2018; Radoi et al. 2014; Smith and Albarghouthi 2016], structural recursion [Farzan et al. 2022; Farzan and Nicolet 2021a], and incrementalization [Liu and Stoller 2003]. These approaches will fail when the output of the original program cannot be directly calculated, for example, when applying D&C to \textit{sndmin} (where the first minimum is required as an auxiliary value). Compared with these approaches, \textit{AutoLifter} supports finding necessary auxiliary values.

**Type- and resource-aware synthesis.** There is another line of work for synthesizing efficient programs, namely type- and resource-aware synthesis [Hu et al. 2021; Knoth et al. 2019]. These approaches use a type system to represent a resource bound, such as the time complexity, and use type-driven program synthesis [Polikarpova et al. 2016] to find programs satisfying the given bound.

Compared with \textit{AutoLifter}, these approaches can deal with more refined efficiency requirements via advanced type systems. However, they suffer from a more serious scalability challenge because they need to synthesize the whole resulting program from the start. As far as we are aware, so far none of these approaches can scale up to applying algorithmic paradigms as our approach can.

**Program synthesis.** Program synthesis is an active field and many synthesizers have been proposed. Here we only discuss the most related approaches.

First, \textit{AutoLifter} addresses the scalability challenge by decomposing lifting problems into simpler subtasks. This decomposition-based framework is common in program synthesis and has been applied to various scenarios. We list some representative approaches in this category as follows.

- **Natural Synthesis** [Qiu and Solar-Lezama 2017] uses loop invariants to decompose a loop synthesis problem into subtasks for pre-loop, in-loop, and after-loop codes, respectively.
- **Myth** [Osera and Zdancewic 2015] and **Synquid** [Polikarpova et al. 2016] use a top-down enumeration scheme to synthesize recursive programs and will utilize type information to decompose certain intermediate synthesis problems into independent subtasks. Our decomposition method, component elimination, is related to the tuple-decomposition method in **Myth**. Both of them are proposed to decompose an unknown program with a
tuple output. In comparison, the method in Myth requires the specification of the unknown program to be input-output examples, while component elimination considers a specification where the input and the output both depend on another unknown program.

- DryadSynth [Huang et al. 2020] proposes a general framework for reconciling inductive and deductive program synthesis. This framework repeatedly applies deductive rules to decompose a synthesis task into subtasks and then solves these subtasks using inductive program synthesis. DryadSynth implements this framework for the domain of conditional integer arithmetic, and AutoLifter can be regarded as an implementation of this framework for solving lifting problems.

- Toshokan [Huang and Qiu 2022] proposes a decomposition method for dealing with complex library calls in component-based program synthesis. It uses library models to decompose a synthesis problem involving library calls into subtasks of library verification and client-code synthesis. This decomposition method cannot be applied to our task because lifting problems do not involve any libraries.

Second, since many algorithms can be regarded as recursive programs (e.g., D&C and single-pass programs), AutoLifter is also related to previous studies on recursive program synthesis [Albarghouthi et al. 2013; Farzan et al. 2022; Farzan and Nicolet 2021a; Lee and Cho 2023; Miltner et al. 2022; Yuan et al. 2023]. However, previous recursive synthesizers cannot be applied to our tasks because of two major differences between recursive program synthesis and the automatic application of algorithmic paradigms.

- The two problems treat input-output examples (or the original program) differently. Recursive program synthesis typically treats input-output examples as a full specification. It requires the synthesized recursive procedure to produce exactly the same outputs as specified in the examples. However, when applying algorithmic paradigms, we often need to introduce auxiliary values as part of the output of the recursive procedure. Sticking to the same outputs would lead to synthesis failures.

- The two problems put different restrictions on the recursion. In recursive program synthesis, the synthesizer can use any recursion that can implement the target functions. In contrast, when applying algorithmic paradigms, the recursion of programs is prescribed by the paradigm, and the problem is how to calculate using the given recursions. Both settings have their respective challenges. In recursive program synthesis, the challenge is to find a proper recursion; and when applying algorithmic paradigms, the given recursion may significantly increase the scale of the resulting program. For example, the D&C program of snadmin (Figures 3 and 4) is much more complex than its single-pass program (Figure 6), though both of them can be regarded as recursive programs.

Besides, our reduction from applying algorithmic paradigms to lifting problems shares the same idea with trace completeness in recursive program synthesis [Albarghouthi et al. 2013]. They both utilize the interpretation of the original program and then reduce the problem of synthesizing recursions to the problem of synthesizing the body of recursions.

At last, AutoLifter is also related to Enum [Alur et al. 2013] and Relish [Wang et al. 2018] because they can also be applied to solve lifting problems. We compare AutoLifter with both of them in our evaluation, and the results demonstrate the better performance of AutoLifter.

11 CONCLUSION

In this paper, we study the problem of applying D&C-like algorithmic paradigms and aim to address the limitation of previous transformation-based approaches which put strict restrictions on the original program. To achieve this goal, we propose a novel approach named AutoLifter that applies
D&C-like paradigms by decomposition instead of by syntax-based transformation. To achieve an effective synthesis, AutoLifter repeatedly applies two decomposition methods, namely component elimination and variable elimination, to decompose an application task into simpler subtasks and derive specifications for different sub-programs of the synthesis target.

To break the dependency among sub-programs, both decomposition methods in AutoLifter use approximate specifications in their first subtasks. We demonstrate that these approximations do not affect the effectiveness of AutoLifter by conducting theoretical analysis and empirical evaluation. In theory, we prove that these approximations will seldom produce unrealizable subtasks when the compressing property holds; in practice, we evaluate AutoLifter on a dataset of 96 tasks and the results show that AutoLifter can solve most of these problems within a short time.

We believe many techniques in this paper are general and can be potentially applied to other tasks. For example, variable elimination may be used to separate the composition of two unknown programs in other relational synthesis problems. Exploring other applications is future work.

The source code of our implementation and the experimental data of our evaluation are available online [Ji et al. 2024].

ACKNOWLEDGEMENT

We thank all anonymous reviewers of this paper for their valuable suggestions on this work. We also thank Sirui Lu and Rastislav Bodík for their insightful feedback on this work and generous help in our evaluation. This work is supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902.

REFERENCES


A APPENDIX: PROOFS AND GUARANTEES

This section provides the proofs for the theorems in this paper (Section 5) and supplies the details on the probabilistic correctness guarantee provided by our verifier (Section 7).

A.1 Proofs for Theorems

**Theorem A.1 (Theorem 5.1).** The result of AutoLifter (Algorithm 1) is valid for the original lifting program if the verifiers of leaf subtasks accept only valid programs for respective subtasks.

**Proof.** The soundness of AutoLifter is directly implied by the two facts below.

- The result of AutoLifter must be valid when the sub-programs synthesized from leaf subtasks are valid because both decomposition methods in AutoLifter are sound by definition (i.e., the merged result is valid for the original task when the sub-results are valid for the subtasks).
- Since leaf subtasks are solved using the CEGIS framework, their results must be valid when the verifiers accept only valid programs.

□

**Theorem A.2 (Theorem 5.4).** Given a lifting problem \( LP(\text{orig}, \text{op}) \) of which the mismatch factor is at least \( t \), the size-limited unrealizable rate of AutoLifter is bounded, as shown below.

\[
\text{unreal}(\text{orig}, \text{op}, \text{lim}_s) \leq 2^w \exp(-t/s_V w), \text{ where } w \triangleq \lim_c \cdot \lim_s
\]

**Proof.** For simplicity, we shall abbreviate our probabilistic model \( \mathcal{M}[(\text{orig}, \text{op})] \) as \( \mathcal{M} \) and interchangeably use a synthesis task as a predicate, where \( \varphi(\text{prog}) \) represents that \( \text{prog} \) is a valid program for task \( \varphi \). Besides, we shall use the following two notations in our proof.

- Let \( \tilde{\varphi} \) be the first \( \text{aux} \) subtask generated from \( \varphi \), of which the specification is shown below.

\[
(\text{orig} \triangle \text{aux}_1)^n \tilde{a} = (\text{orig} \triangle \text{aux}_1)^n \tilde{a} \rightarrow \text{orig} (\text{op} (c, \tilde{a})) = \text{orig} (\text{op} (c, \tilde{a}'))
\]

- Let \( \mathcal{A}(\varphi) \) be the set of auxiliary programs that can lead to a valid solution of \( \varphi \) with a size no larger than \( \text{lim}_s \), defined as below. Using this notation, the condition in the size-limited unrealizable rate (Definition 5.2) can be restated as \( |\mathcal{A}(\varphi)| > 0 \).

\[
\mathcal{A}(\varphi) \triangleq \{ \text{aux} | \exists \text{comb}, \varphi(\text{aux; comb}) \land \text{size(aux; comb)} \leq \text{lim}_s \}
\]

**Step 1: a sufficient condition.** Any program in \( \mathcal{A}(\varphi) \) is valid for \( \tilde{\varphi} \) by definition. Furthermore, AutoLifter will not generate unrealizable subtasks when a program in \( \mathcal{A}(\varphi) \) is synthesized from \( \tilde{\varphi} \). Specifically, when such a program is synthesized, AutoLifter will only generate 3 subtasks after \( \tilde{\varphi} \), as shown below.

1. The first \( \text{comb} \) subtask targets at a combinator for the output of \( \text{orig} \). It is realizable because there is always a valid combinator for any program in \( \mathcal{A}(\varphi) \).
2. The second \( \text{aux} \) subtask targets at an auxiliary program for the output of the synthesis result of \( \tilde{\varphi} \). The result of this subtask must be \( \text{null} \) as (1) it is valid by the definition of \( \mathcal{A}_{\text{aux}} \), and (2) it is the first choice of the corresponding synthesizer \( S_{\text{aux}} \) (Algorithm 4).
3. The second \( \text{comb} \) subtask targets at a combinator for the output of the auxiliary program, which is also realizable by the definition \( \mathcal{A}_{\text{aux}} \).

Therefore, the event that the synthesis result of \( \tilde{\varphi} \) is in \( \mathcal{A}(\varphi) \) is a sufficient condition for the success of AutoLifter. As a result, we know that the size-limited unrealizable rate of AutoLifter is bounded by the probability below.

\[
\Pr_{\varphi \sim \mathcal{M}} \left[ S_{\text{aux}}(\tilde{\varphi}) \notin \mathcal{A}(\varphi) \mid |\mathcal{A}(\varphi)| > 0 \right]
\]
This probability is difficult for direct analysis because it involves second-order quantification (in the definition of \( \hat{A}(\varphi) \)) and the concrete behavior of a synthesizer, which can be very complex. Therefore, we conduct a series of derivations to eliminate these difficult parts from the probability.

**Step 2: eliminating left-side \( \hat{A}(\varphi) \).** The conditional probability can be transformed as follows.

\[
\Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin \hat{A}(\varphi) \mid |\hat{A}(\varphi)| > 0 \right]
\]

\[
= \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin \hat{A}(\varphi) \right] / \Pr_{\varphi \sim M} \left[ |\hat{A}(\varphi)| > 0 \right]
\]

\[
= \sum_{P \subseteq L_{aux}} [|P| > 0] \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin P \wedge \hat{A}(\varphi) = P \right] / \sum_{P \subseteq L_{aux}} [|P| > 0] \Pr_{\varphi \sim M} \left[ \hat{A}(\varphi) = P \right]
\]

\[
\leq \sum_{P \subseteq L_{aux}} [|P| > 0] \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin P \wedge \hat{A}(\varphi) = P \right] + (|P| - 1) \Pr_{\varphi \sim M} \left[ \hat{A}(\varphi) = P \right]
\]

(25)

The inequality of \((a + c)/(b + c) \geq a/b\) (when \(a, b > 0, c \geq 0, a < b\)) is used in the last step.

Let us consider the claim below under the premise that \(|P| > 0\).

\[
\Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin P \wedge \hat{A}(\varphi) = P \right] + (|P| - 1) \Pr_{\varphi \sim M} \left[ \hat{A}(\varphi) = P \right]
\]

\[
= \sum_{aux^* \in P} \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin aux^* \wedge \hat{A}(\varphi) = P \right]
\]

To prove this claim, let \( \varphi \) be any task satisfying \( \hat{A}(\varphi) = P \). There are two cases on \( S_{aux}(\tilde{\varphi}) \).

- When \( S_{aux}(\tilde{\varphi}) \notin P \), the probability of \( \varphi \) contributes to both sides for \(|P|\) times.
- Otherwise, the probability of \( \varphi \) contributes to both sides for \(|P| - 1\) times.

Therefore, the two probabilities involved in this claim must be the same.

Note that the size of any program in \( \hat{A}(\varphi) \) is no larger than \( \text{lim}_s \) by the definition of \( \hat{A}(\varphi) \). By applying the claim to Formula 26, we further perform the derivation below, where \( L_{\leq \text{lim}_s} \) denotes the subspace of \( L_{aux} \) including only those programs of which the size is no larger than \( \text{lim}_s \).

\[
\text{Formula 26} = \sum_{P \subseteq L_{aux}} \sum_{aux^* \in P} \sum_{\varphi \in M} \left[ S_{aux}(\tilde{\varphi}) \notin aux^* \wedge \hat{A}(\varphi) = P \right] / \sum_{P \subseteq L_{aux}} \sum_{aux^* \in P} \Pr_{\varphi \sim M} \left[ \hat{A}(\varphi) = P \right]
\]

\[
= \sum_{aux^* \in L_{\leq \text{lim}_s}} \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin aux^* \wedge aux^* \in \hat{A}(\varphi) \right] / \sum_{aux^* \in L_{\leq \text{lim}_s}} \Pr_{\varphi \sim M} \left[ aux^* \in \hat{A}(\varphi) \right]
\]

\[
\leq \max_{aux^* \in L_{\leq \text{lim}_s}} \left( \sum_{\varphi \sim M} \Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \notin aux^* \wedge aux^* \in \hat{A}(\varphi) \right] \right) / \Pr_{\varphi \sim M} \left[ aux^* \in \hat{A}(\varphi) \right]
\]

(27)

\[
\text{Formula 27} = \max_{aux^* \in L_{\leq \text{lim}_s}} \left[ S_{aux}(\tilde{\varphi}) \notin aux^* \mid aux^* \in \hat{A}(\varphi) \right]
\]

(28)

Formula 27 is obtained via the following inequality (where 0/0 is defined as 0).

\[
\forall a_i, b_i \geq 0, \sum_{i=1}^n a_i / \sum_{i=1}^n b_i \leq \max_{i=1}^n (a_i/b_i)
\]

So far, the left-side \( \hat{A}(\varphi) \) in the sufficient condition (Formula 24) has been eliminated.

**Step 3: eliminating the output of \( S_{aux} \).** A worth noting property of \( S_{aux} \) (Algorithm 4) is that, given a task that has a solution with a size no larger than \( \text{lim}_c \), the size of its synthesis result is no larger than \( \text{lim}_c \text{lim}_s \). First, the result of \( S_{aux} \) includes at most \( \text{lim}_c \) components. Second, the size of
each component must be no larger than $lim_s$, otherwise, the smallest solution (which is no larger than $lim_a$) will be found by OE and be synthesized instead.

Therefore, “any program in $L \leq \text{lim}_a \text{lim}_s$ is invalid for $\phi$ except $aux^*$” forms a sufficient condition of $S_{aux}(\phi) = aux^*$. Then, the conditional probability in Formula 28 can be bounded as follows.

$$
\Pr_{\phi \sim \mathcal{M}} [S_{aux}(\phi) \neq aux^* \mid aux^* \in \mathcal{A}(\phi)]
\leq \Pr_{\phi \sim \mathcal{M}} [\exists aux \in L \leq \text{lim}_a \text{lim}_s, aux \neq aux^* \land \phi(aux) \mid aux^* \in \mathcal{A}(\phi)]
\leq \sum_{aux \in L \leq \text{lim}_a \text{lim}_s} [aux \neq aux^*] \Pr_{\phi \sim \mathcal{M}} [\phi(aux) \mid aux^* \in \mathcal{A}(\phi)]
\leq 2^{\text{lim}_a \text{lim}_s} \max_{aux \in L \leq \text{lim}_a \text{lim}_s} [aux \neq aux^*] \Pr_{\phi \sim \mathcal{M}} [\phi(aux) \mid aux^* \in \mathcal{A}(\phi)]
$$

(29)

The last step uses the fact that the number of programs whose size is no larger than $k$ is at most $2^k$. So far, the concrete output of $S_{aux}$ has been eliminated.

**Step 4: eliminating the condition in the probability.** The probability in Formula 29 can be transformed as follows by unfolding the definition of $\mathcal{A}(\phi)$.

$$
\Pr_{\phi \sim \mathcal{M}} [\phi(aux) \mid aux^* \in \mathcal{A}(\phi)] = \Pr_{\phi \sim \mathcal{M}} [\phi(aux) \mid \exists \text{comb}, \phi(aux^*, \text{comb}) \land \text{size}(aux^*, \text{comb}) \leq \text{lim}_s]
$$

Recall that our probabilistic model $\mathcal{M}$ is parameterized by the semantics of orig and op, and its randomness comes only from the semantics of programs in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$. In the probability above, the outcome $\phi(aux)$ is determined only by $aux$, while the condition $aux^* \in \mathcal{A}(\phi)$ is determined by $aux^*$ and programs in $\mathcal{L}_{comb}$. Therefore, the two events in this conditional probability are independent, making it safe to directly ignore the condition, as shown below.

$$
\Pr_{\phi \sim \mathcal{M}} [\phi(aux) \mid aux^* \in \mathcal{A}(\phi)] = \Pr_{\phi \sim \mathcal{M}} [\phi(aux)]
$$

**Step 5: bounding the probability of $\phi(aux)$ using the mismatch factor.** Let us first unfold and transform the above probability $\Pr[\phi(aux)]$ as follows.

$$
\Pr_{aux} [\phi(aux)]
= \Pr_{aux} [(\text{orig} \land aux)^n \overline{a} = (\text{orig} \land aux)^n \overline{a'} \rightarrow \text{orig} (op (c, \overline{a})) = \text{orig} (op (c, \overline{a'}))]
= \Pr_{aux} [(\text{orig}^n \overline{a} = \text{orig}^n \overline{a'} \land \text{orig} (op (c, \overline{a})) \neq \text{orig} (op (c, \overline{a'}))) \rightarrow aux^n \overline{a} \neq aux^n \overline{a'}]
$$

(30)

Suppose the mismatch factor of $(\text{orig}, op)$ is at least $t$, that means, there are $t$ pairs of orig inputs $(\overline{a}_i, \overline{a}'_i)$ such that (1) every pair satisfies the premise of the event in Formula 30, as shown below,

$$
\exists c, (\text{orig}^n \overline{a}_i = \text{orig}^n \overline{a}'_i \land \text{orig} (op (c, \overline{a}_i)) \neq \text{orig} (op (c, \overline{a}'_i)))
$$

and (2) all components involved in these pairs ($2tn$ in total) are different.

The event in Formula 30 is satisfied only when $aux^n$ outputs differently on all these $t$ pairs of inputs. Using this fact, the target probability can be bounded as follows.

$$
\Pr_{aux} [\phi(aux)] \leq \Pr_{aux} \left[ \bigwedge_{i=1}^{t} aux^n \overline{a}_i \neq aux^n \overline{a}'_i \right] \leq \prod_{i=1}^{t} \Pr_{aux} [aux^n \overline{a}_i \neq aux^n \overline{a}'_i] \leq (1 - \text{pow}(s_v, -\text{lim}_c \text{lim}_s))^t \leq \exp (-t/\text{pow}(s_v, n\text{lim}_c \text{lim}_s))
$$

We supply some details on the above derivation as follows.
The second step uses the premise that all components involved in \( \overline{a_i} \) and \( \overline{a'_i} \) are different. This fact implies that events \( aux^n \overline{a_i} \neq aux^n \overline{a'_i} \) are totally independent.

The third step uses the fact that the size of \( aux \) is no longer than \( lim_v \cdot lim_s \). Under this condition, \( aux \) can only output at most \( lim_v \cdot lim_s \) auxiliary values, and thus the size of the output domain of \( aux \) is no longer than \( pow(s_V, lim_v \cdot lim_s) \).

**Step 6: Summary.** Let us now sum up the previous sub-results and then prove the target theorem. First, the inequality below is obtained by combining Steps 4 and 5.

\[
\Pr_{\varphi \sim M} \left[ \tilde{\varphi}(aux) \mid aux^* \in \Lambda(\varphi) \right] \leq \exp(-t/s_V^{n-w}), \text{ where } w \triangleq lim_v \cdot lim_s
\]

Second, the inequality below is obtained by further combining Step 3.

\[
\Pr_{\varphi \sim M} \left[ S_{aux}(\tilde{\varphi}) \neq aux^* \mid aux^* \in \Lambda(\varphi) \right] \leq 2^w \exp(-t/s_V^{n-w})
\]

At last, we know the size-limited unrealizable rate of Autolifter is bounded by the formula below after further combining Steps 1 and 2, which is exactly the target theorem.

\[
2^w \exp(-t/s_V^{n-w}), \text{ where } w \triangleq lim_v \cdot lim_s
\]

Before proving Theorem 5.6, we shall first introduce and prove the following lemma. It shows that the mismatch factor of a random lifting problem is almost surely large.

**Lemma A.3.** When there are at least two values (i.e., \( s_V \geq 1 \)), for any constant \( \epsilon > 0 \), there always exists a constant \( \delta \) such that the probability for the mismatch factor of a random lifting problem to be smaller than \( \delta \cdot s_A/(n \cdot s_V^{n+1}) \) does not exceed \( \epsilon \), as shown below.

\[
\forall \epsilon > 0, \exists \delta, \forall n, A, V, \Pr_{\text{orig,op}} \left[ \text{the mismatch factor of } LP(\text{orig, op}) < [\delta \cdot s_A/(n \cdot s_V^{n+1})] \right] \leq \epsilon
\]

where \( \text{orig} \) is a random function with type \( A \rightarrow V \), \( \text{op} \) is a random function with type \( V \times A^n \rightarrow A \), \( s_A \) and \( s_V \) are the numbers of values in types \( A \) and \( V \), respectively, and it is assumed that \( s_V > 1 \).

**Proof.** In this proof, we only need to consider the case where \( s_A/(n \cdot s_V^{n+1}) \) is large enough. To see this point, let \( t \) be an arbitrary threshold that may depend on \( \epsilon \). By taking \( \delta \) as a value smaller than \( 1/t \), the target inequality will be satisfied in cases where \( s_A/(n \cdot s_V^{n+1}) \leq t \), as shown below.

\[
\Pr_{\text{orig,op}} \left[ \text{the mismatch factor of } LP(\text{orig, op}) < [\delta \cdot s_A/(n \cdot s_V^{n+1})] \right] = \Pr_{\text{orig,op}} \left[ \text{the mismatch factor of } LP(\text{orig, op}) < 0 \right] = 0 \leq \epsilon
\]

Therefore, in the remainder of this proof, we shall focus on the cases where \( s_A/(n \cdot s_V^{n+1}) \) is larger than a threshold \( t \). Note that in these cases, \( s_A \) must also be large because \( s_V \) is at least 2.

As the first step, we divide values in type \( A \) into subsets \( A_1 \) and \( A_2 \), whose sizes are \( \alpha s_A \) and \((1 - \alpha)s_A \) values for some constant \( \alpha \), respectively. This construction ensures that the outputs of \( \text{orig} \) on \( A_1 \) and \( A_2 \) are independent.

Second, we prove that with a high probability, \( \text{orig} \) will not map too many values in \( A_2 \) to the same value. Let \( \text{num}(v) \) for \( v \in V \) be the number of values in \( A_2 \) on which the outputs of \( \text{orig} \) is \( v \).
We bound the probability of \( \exists v \in V, \text{num}(v) \geq 2|A_2|/3 \) (denoted as event \( \mathcal{E} \)) as follows.

\[
\Pr[\mathcal{E}] \leq \sum_{v \in V} \Pr[\text{num}(v) > 2|A_2|/3] \\
\leq \sum_{v \in V} \Pr[\text{num}(v) > (1 + 1/3)\mathbb{E}[\text{num}(v)]] \\
\leq s_v \cdot \exp\left(-\frac{(1 - \alpha)s_A}{21s_v}\right)
\]

Here the first step uses the union bound, the second step uses the facts that \( \mathbb{E}[\text{num}(v)] \leq |A_2|/s_o \) and \( s_v \geq 2 \), and the last step uses the Chernoff bound.

Third, we assume that \( \mathcal{E} \) does not happen and then construct a valid sequence of input pairs for the mismatch factor using only values in \( A_1 \). Concretely, values in \( A_1 \) can be arranged into a sequence \( S \) including \( m = |A_1|/(2n) \) independent input pairs in \( A^n \times A^n \). Then, we call an input pair \((\overline{a}, \overline{a'})\) as valid if it satisfies the conditions below for a given value \( c \in C \).

\[
\text{orig} \, \overline{a} = \text{orig} \, \overline{a'} \quad \text{orig} \, (op \, (c, \overline{a})) \neq \text{orig} \, (op \, (c, \overline{a'})) \quad op \, (c, \overline{a}) \in A_2 \quad op \, (c, \overline{a'}) \in A_2
\]

Let \( S' \) be the sub-sequence of \( S \) that includes only valid pairs. The left two conditions above ensure that \( S' \) is a valid sequence for the mismatch factor. Therefore, \( |S'| \) provides a lower bound on the mismatch factor, and the task remaining is to prove that \( |S'| \) is large with a high probability.

For an input pair \((\overline{a}, \overline{a'})\) in \( S \), the probability for it to satisfy the first condition is \( s_v^n \), and the probability for it to satisfy the other three conditions is at least \( (1 - \alpha)^2/3 \) when event \( \mathcal{E} \) does not happen. These two probabilities are independent because the outputs of \( op \), the outputs of \( \text{orig} \) on \( A_1 \), and the outputs of \( \text{orig} \) on \( A_2 \) are all independent. Therefore, the probability for a pair in \( S \) to be valid is at least \( (1 - \alpha)^2/3 \cdot s_v^n \).

Now let us bound the length of sub-sequence \( S' \). On the one hand, the expectation of \( |S'| \) is \( (1 - \alpha)^2/3 \cdot s_v^n \cdot m \), which is no smaller than \( \gamma \cdot s_A/(n \cdot s_v^n) \) for some constant \( \gamma < (1 - \alpha)^2/6 \). On the other hand, it is easy to verify that the probabilities for each pair in \( S \) to be valid are independent when the outputs of \( \text{orig} \) on \( A_2 \) are fixed. Therefore, the Chernoff bound can be applied to provide a probabilistic lower bound for \( |S'| \), as shown below, where \( \tau \) is an arbitrary constant in \( (0, 1) \).

\[
\Pr\left[|S'| \leq (1 - \tau) \cdot \gamma \cdot s_A/(n \cdot s_v^n) \right] \leq \Pr\left[|S'| \leq (1 - \tau)\mathbb{E}[|S'|] \right] \leq \exp\left(-\frac{\tau^2}{2} \cdot \gamma \cdot \frac{s_A}{n \cdot s_v^n}\right)
\]

By combining the above results, we can bound the mismatch factor as follows.

\[
\Pr[\text{the mismatch factor of LP}(\text{orig}, op) \leq (1 - \tau) \cdot \gamma \cdot s_A/s_v^n] \\
\leq \Pr\left[|S'| \leq (1 - \tau) \cdot \gamma \cdot s_A/(n \cdot s_v^n) \right] + \Pr[\mathcal{E}] \\
\leq \exp\left(-\frac{\tau^2}{2} \cdot \gamma \cdot \frac{s_A}{n \cdot s_v^n}\right) + s_v \cdot \exp\left(-\frac{(1 - \alpha)s_A}{21s_v}\right)
\]

Since we have assumed that \( s_A/(n \cdot s_v^{n+1}) \) is no smaller than some value \( t \), the value of \( s_A/(n \cdot s_v^n) \) is no smaller than \( t \), and the value of \( s_A/s_v \) is no smaller than \( t \cdot s_v \). Therefore, the formula above can...
be further simplified as below, resulting in the target lemma.

$$\leq \exp\left(-\frac{r^2}{2} \cdot y \cdot t\right) + s_V \cdot \exp\left(-\frac{(1 - \alpha)}{21} \cdot s_V \cdot t\right)$$

$$\leq \exp\left(-\frac{r^2}{2} \cdot y \cdot t\right) + \exp\left(-\frac{(1 - \alpha)}{21} \cdot t\right) \text{ when } t > \frac{210}{1 - \alpha}$$

$$\leq \epsilon \text{ when } t > k \cdot \ln(1/\epsilon) \text{ for a large enough constant } k$$

THEOREM A.4 (THEOREM 5.6). Consider the size-limited unrealizable rate of AutoLifter on a random lifting problem. When there are at least two values (i.e., \(s_V > 1\)), for any constant \(\epsilon > 0\), the probability for this rate to exceed \(\epsilon\) tends to 0 when \(s_A/s_V^{\omega'}\) tends to \(\infty\), where \(\omega' \doteq n \cdot \lim_c \cdot \lim_s + n + 1\).

PROOF. To prove this theorem, we need to prove that for any constants \(\epsilon, \epsilon' > 0\), the probability for the unrealizable rate to exceed \(\epsilon\) is at most \(\epsilon'\) when \(s_A/s_V^{\omega'}\) is large enough, as shown below.

$$\Pr_{\text{orig,op}}\left[\text{unreal(\text{orig, op}, \lim_s) > } \epsilon\right] \leq \epsilon'$$

By Lemma A.3, there exists a constant \(\delta\) such that with a probability of at least \(1 - \epsilon'\), the mismatch factor on a random lifting problem will be at least \([\delta \cdot s_A/(n \cdot s_V^{\omega'})]\), as shown below.

$$\Pr_{\text{orig,op}}\left[\text{the mismatch factor of LP(\text{orig, op})} \geq [\delta \cdot s_A/(n \cdot s_V^{\omega'})]\right] \geq 1 - \epsilon'$$

When the event in the above probability happens (denoted as event \(\mathcal{E}\)), the size-limited unrealizable rate is bounded by Theorem 5.4, as shown below.

$$\mathcal{E} \rightarrow \text{unreal(\text{orig, op}, \lim_s) \leq } 2^w \exp\left(-[\delta \cdot s_A/(n \cdot s_V^{\omega'})]/s_V^{\omega'-w}\right), \text{where } w \doteq \lim_c \cdot \lim_s$$

When \(s_A/s_V^{\omega'}\) is larger than a threshold \(k\), the above formula can be simplified as follows.

$$2^w \exp\left(-[\delta \cdot s_A/(n \cdot s_V^{\omega'})]/s_V^{\omega'-w}\right) \leq 2^w \exp(-\delta \cdot k/n) \leq \epsilon \text{ when } k \text{ is large enough}$$

Therefore, we know the following inequality holds when \(s_A/s_V^{\omega'}\) is large enough. This result directly implies the target conclusion.

$$\Pr_{\text{orig,op}}\left[\text{unreal(\text{orig, op}, \lim_s) > } \epsilon\right] \leq \Pr_{\text{orig,op}}\left[\text{unreal(\text{orig, op}, \lim_s) > } \epsilon \mid \mathcal{E}\right] + \Pr_{\text{orig,op}}\left[\neg \mathcal{E}\right]$$

$$< \epsilon' \text{ when } s_A/s_V^{\omega'} \text{ is large enough}$$

THEOREM A.5 (THEOREM 5.8). Given an example-based task for the auxiliary program, let \(\text{aux}^*\) be the synthesis result of \(\mathcal{S}_{\text{aux}}\). Then, any sub-program of \(\text{aux}^*\) must not be valid for the given task.

PROOF. Assume that there is a strict sub-program of \(\text{aux}^*\) (denoted as \(\text{aux}\)) that is also valid for the example-based task. Then there are two possible cases.

**Case 1:** \(\text{aux}\) is strictly included in a component of \(\text{aux}^*\). Let \(\text{comp}\) be the corresponding component. Since OE enumerates programs strictly from small to large, it must return \(\text{aux}\) as a component before \(\text{comp}\). Therefore, \(\text{aux}\) will be considered by \(\mathcal{S}_{\text{aux}}\) before \(\text{aux}^*\), so \(\text{aux}\) should be the synthesis result of \(\mathcal{S}_{\text{aux}}\) instead, leading to a conflict.

**Case 2:** \(\text{aux}\) is not strictly included in any component of \(\text{aux}^*\). Since \(\text{aux}^*\) is formed as a tuple of components (i.e., in the form of \(\text{comp}_1 \triangleleft \ldots \triangleleft \text{comp}_k\)), as a sub-program of \(\text{aux}^*\), \(\text{aux}\) must be a tuple of several components in \(\text{aux}^*\). So \(\text{aux}\) must be considered by \(\mathcal{S}_{\text{aux}}\) before \(\text{aux}^*\) since the
top-level combination enumerates the number of components from small to large. Consequently, aux should be the synthesis result of \( S_{aux} \) instead, leading to a conflict.

In summary, the assumption never holds, and thus the target theorem is obtained. □

### A.2 Example of Analyzing the Mismatch Factor

In Section 2.1, we have discussed a lifting problem about applying D&C to sndmin, where the original program orig calculates sndmin and the operator op concatenates two given lists. Now, we are going to prove a lower bound for the mismatch factor of this lifting problem. For simplicity, in the discussion below, we assume each input list includes up to \( l \) integers in the range \([1, sv]\).

To get a lower bound of the mismatch factor, we need to find a sequence of input pairs satisfying the two conditions in Definition 5.3. Specifically, in this example, we need to find a sequence of 4-list tuples such that (1) every tuple \( ((xs_L, xs_R), (xs'_L, xs'_R)) \) in this sequence satisfies the formula below, and (2) every list is used at most once in this sequence.

\[
(sndmin xs_L, sndmin xs_R) = (sndmin xs'_L, sndmin xs'_R) \quad \forall \quad sndmin (xs_L + xs_R) \neq sndmin (xs'_L + xs'_R)
\]  

(31)

We can construct such a sequence from those lists formed by \( l - 2 \) integers in range \([3, sv]\). Specifically, for every such list \( ys \), we construct a tuple \( ((xs_L, xs_R), (xs'_L, xs'_R)) \) as shown below\(^{11}\).

\[
\begin{align*}
xs_L &\triangleq [1, 2] \quad xs_R \triangleq [1, 3] \\
xs'_L &\triangleq [2, 2] \quad xs'_R \triangleq [2, 3]
\end{align*}
\]

This construction results in a sequence of length \((sv - 2)^{l-2}\), the number of different \( ys \). We can verify that this sequence indeed satisfies the two conditions in Definition 5.3.

- **For the first condition**, Formula 31 is satisfied by all tuples because the outputs of \( sndmin \) on \( xs_L \) and \( xs'_L \) are always 2, the outputs on \( xs_R \) and \( xs'_R \) are always 3, but the outputs on \( xs_L + xs_R \) and \( xs'_L + xs'_R \) are always different, which are 1 and 2, respectively.
- **For the second condition**, no two lists in this sequence are the same because (1) no two lists in the same tuple can be the same as their first two elements must be different, and (2) no two lists in different tuples can be the same as their last \( l - 2 \) elements must be different.

Therefore, the mismatch factor of the lifting problem in Section 2.1 is at least \((sv - 2)^{l-2}\). By combining this lower bound with Theorem 5.4, we can get the conclusion that the size-limited unrealizable rate of this lifting problem tends to 0 when the length of lists (i.e., \( l \)) tends to \( \infty \).

### A.3 Probabilistic Correctness Guarantee of Our Verifier

Our implementation of AutoLifter includes a testing procedure as a part of the verifier (Section 7). In the \( i \)-th CEGIS iteration, it tests the candidate program using \( 10^4 \times i \) random examples generated from a pre-defined distribution. This testing procedure provides a guarantee that the probability for the error rate of the synthesis result to be more than \( 10^{-3} \) is at most \( 4.55 \times 10^{-5} \).

To prove this guarantee, let us first introduce some necessary notations. Let \( n \) be the number of examples used in the first CEGIS iteration \((10^4 \text{ in our implementation})\) and \( \delta \) be the tolerable error rate in the probabilistic guarantee \((10^{-3} \text{ in the above claim})\). Then, let \( E \) be the event that the error rate of the synthesis result is more than \( \delta \), and let \( E_i \) be the event that a program with an error rate larger than \( \delta \) is accepted by the verifier in the \( i \)-th CEGIS iteration.

To get a probabilistic guarantee, our target is to derive an upper bound for \( \Pr[E] \). By the definition, \( E \) happens only when some event \( E_i \) happens, and \( E_i \) happens only when a program with an error rate larger than \( \delta \) is accepted by the verifier in the \( i \)-th CEGIS iteration.

\(^{11}\)In this construction, we assume the value of \( sv \) is at least 3.
rate larger than $\delta$ passes $n \cdot i$ random examples, of which the probability is no larger than $(1 - \delta)^{n-i}$. Therefore, the following inequality holds.

$$\Pr[\mathcal{E}] \leq \sum_{i=1}^{+\infty} \Pr[\mathcal{E}_i] \leq \sum_{i=1}^{+\infty} (1 - \delta)^{n-i} \leq \sum_{i=1}^{+\infty} \exp(-\delta \cdot n \cdot i) = w/(1 - w) \text{ for } w \triangleq \exp(-\delta \cdot n)$$

When $(n, \delta)$ is set to $(10^4, 10^{-3})$, the value of $w$ is $\exp(-10) \approx 4.54 \times 10^{-5}$, and thus the upper bound of $\Pr[\mathcal{E}]$ is $w/(1 - w) < 4.55 \times 10^{-5}$. Therefore, the probability for the error rate of the synthesis result to be more than $10^{-3}$ is at most $4.55 \times 10^{-5}$.

\section*{B Appendix: Algorithmic Paradigms}

In this section, we supply the details on the remaining two paradigms of longest segment problems and the paradigm of segment trees. For each paradigm, we introduce three aspects in order: (1) its procedure, (2) its reduction to lifting problems, and (3) the time complexity under the efficiency condition (Section 6.1), i.e., the efficiency guarantee provided by \textit{AutoLifter}.

\subsection*{B.1 The first algorithmic paradigm for the longest segment problem}

Recall that a longest segment problem is specified by a predicate $p$ on lists. Given a list $xs$, this problem asks for the length of the longest segment in $xs$ satisfying the predicate.

\textbf{Procedure.} The first paradigm proposed by Zantema [1992] aims at the cases where predicate $p$ is \textit{predict-closed} and \textit{overlap-closed}, defined as follows.

- \textit{prefix-closed} means that for any list satisfying the predicate, all its prefixes must also satisfy the predicate, i.e., $p\ (xs \#\ ys) \rightarrow p\ xs$.

- \textit{overlap-closed} means that for any two overlapped segments satisfying the predicate, their join must also satisfy the predicate, i.e.,

$$(\text{length } ys > 0 \land p\ (xs \#\ ys) \land p\ (ys \#\ zs)) \rightarrow p\ (xs + ys + zs)$$

This paradigm considers all prefixes of the input list $xs$ in order and calculates the longest suffix satisfying $p$ for each of them. Let $ls(pref)$ be the longest suffix of $pref$ satisfying predicate $p$. For two consecutive prefixes $pref_1$ and $pref_2 = pref_1 + [v]$, when $p$ is both prefix-closed and overlap-closed, $ls(pref_2)$ must be one of $ls(pref_1) + [v], [v]$, and $[\ ]$.

Figure 19 shows a template of this paradigm (in C-like syntax), where $A$ and $n$ denote the input list and the length of the input list respectively. This template calculates the longest valid suffix for each prefix of $A$, stores its length (Line 6), and auxiliary values on the longest valid suffix as $info$ (Line 9). When a new element is considered, $lsp$ verifies whether $ls(A\{0 \ldots i - 1\}) + A_i, A_i, [\ ]$ are valid, and picks the first valid one among them (Lines 9-18). In this procedure, combinator $comb$ is used to quickly update $info$ and verify whether $ls(A\{0 \ldots i - 1\}) + A_i$ is valid (Line 9).

\textbf{Reduction to the lifting problem.} To apply this paradigm, we need to find a combinator $comb$ and an auxiliary program $aux$ satisfying the specification below, where $comb$ updates whether a segment is valid after a new element is appended, and $aux$ provides necessary auxiliary values.

$$(p \land aux)\ (xs + [v]) = comb\ (v, (p \land aux)\ xs)$$

This task can be regarded as a lifting problem $LP(p, op)$ where $op\ (v, (xs)) \triangleq xs + [v]$.

\textbf{Time complexity.} The bottleneck here is $O(n)$ invocations of $comb$ in the loop. Therefore, under the efficiency condition, the resulting program will run in $O(n)$ time on a list of length $n$. 

Decomposition-Based Synthesis for Applying D&C-Like Algorithmic Paradigms

```c
struct Info {
    bool is_valid;
    // Variables representing the output of aux.
};

int lsp(int* A, int n){
    int res = 0, len = 0;
    Info info = {/*p [], aux []*/};
    for (int i = 0; i < n; ++i) {
        info = /*comb (A[i], info)*/;
        if (!info.is_valid) {
            info = {/*p [A[i]], aux [A[i]]*/};
            if (info.is_valid) {
                len = 1;
            } else {
                len = 0;
            }
        } else {
            len += 1;
        }
        res = max(res, len);
    }
    return res;
}
```

Fig. 19. The template of the first paradigm for the longest segment problem.

### B.2 The third algorithmic paradigm for the longest segment problem

**Procedure.** This paradigm does not have any requirement on the $p$ and is based on a technique named *segment partition*. Given list $A[1 \ldots n]$, its segment partition is a series of consecutive segments $(r_0 = 0, r_1], (r_1, r_2], \ldots, (r_{k-1}, r_k = n]$ satisfying (1) $\forall i \in [1, k], \forall j \in (r_{i-1}, r_i), A_j > A_{r_i}$, and (2) $\forall i \in [2, k], A_{r_i-1} \leq A_{r_i}$. This paradigm first generates the segment partition of the given list and then gets the result by merging the information of all segments in the partition.

Figure 20 shows a template of this paradigm, which runs in two steps. In the first step, it constructs a segment partition of the whole list (Lines 8-15). Starting from the empty list, it considers each element in the list, updates the segment partition, and then gets the partition of the whole list when all elements are considered. In this procedure, several variables are used.

- $\text{num}$ represents the number of segments in the current partition (Line 7).
- $\text{rpos}[i]$ represents the index of the right end the $i$th segment in the partition (Line 3).
- $\text{info}[i]$ records the function value of $\text{orig}$ (i.e., the length of the longest valid segment) and $\text{aux}$ on the content of the $i$th segment.

When a new element is inserted, the template merges the last several segments via combinator $\text{comb}$ to ensure that the remaining segments form a partition of the current prefix (Lines 9-14).

In this second step, after the whole segment partition is obtained, the template merges these segments (Lines 16-20) using $\text{comb}$ again and thus gets the result of the whole list (Line 21).

**Reduction to the lifting problem.** In this template, combinator $\text{comb}$ is invoked on an element $v$ and the cared information on two lists $x_{SL}, x_{SR}$, and its task is to return the cared information on $x_{SL} \uplus [v] \uplus x_{SR}$. By the definition of the segment partition, all these invocations ensure that $v$ is the leftmost maximum in $x_{SL} \uplus [v] \uplus x_{SR}$, i.e., $\min x_{SL} > v$ and $\min x_{SR} \geq v$. 
```c
struct Info{
    int res; // Variable representing the output of orig
    // Variables representing the output of aux
} info[N];
int rpos[N];

int solve(int *A, int n) {
    int num = 0;
    for (int i = 0; i < n; i++) {
        Info now = {/*orig [], aux []*/};
        while (num > 0 && A[rpos[num]] > A[i]) {
            now = /*comb (A[rpos[num]], (info[num], now))*/;
            --num;
        }
        num++; rpos[num]=i; info[num]=now;
    }
    Info now = {/*orig [], aux []*/};
    for (int i = num; i > 0; i--) {
        now = /*comb (A[rpos[i]], (info[i], now))*/;
    }
    return now.res;
}
```

Fig. 20. The template of the third paradigm for the longest segment problem.

Therefore, to apply this paradigm, we need to find a combinator \( \text{comb} \) and an auxiliary program \( \text{aux} \) satisfying the following specification.

\[
\begin{align*}
\min \ x_{SL} &> v \land \min \ x_{SR} \geq v \
(\text{orig} \triangle \text{aux})(x_{SL} \mathbin{\oplus} [v] \mathbin{\oplus} x_{SR}) &= \text{comb}(v, (\text{orig} \triangle \text{aux}) x_{SL}, (\text{orig} \triangle \text{aux}) x_{SR})
\end{align*}
\]

This task can be reduced to a lifting problem \( \text{LP}(\text{orig}, \text{op}) \), where \( \text{op} \) is defined as follows and the premise above is eliminated by setting the corresponding outputs of \( \text{op} \) to a dummy list.

\[
\text{op}(v, (x_{SL}, x_{SR})) \triangleq \begin{cases} 
    x_{SL} \mathbin{\oplus} [v] \mathbin{\oplus} x_{SR} & \min \ x_{SL} > v \land \min \ x_{SR} \geq v \\
    [] & \text{otherwise}
\end{cases}
\]

**Time complexity.** Under the efficiency condition, the synthesized program runs in \( O(n) \) time on a list of length \( n \) because its bottleneck is \( O(n) \) invocations of \( \text{comb} \).

### B.3 Segment Tree

The segment tree aims at the problem of Range Update and Range Query [Bentley 1977a], a classical data structure problem. Given an initial list \( x_s \), a query program \( h \), and an update program \( u \), the task is to process a series of operations in order. There are two types of operations.

- Range update \( (U, a, l, r) \): set the value of \( x_s \) to \( u(a, x_s) \) for each \( i \in [l, r] \).
- Range query \( (Q, l, r) \): calculate and output the value of \( h \) \( [x_{S_l}, \ldots, x_{S_r}] \).

The segment tree requires the semantics of the update operator \( u \) to form a monoid, that is, there exists a constant \( a_0 \) and an operator \( \oplus \) satisfying the following conditions.

\[
\forall w, u(a_0, w) = w \quad \forall a_1, \forall a_2, w, u(a_1, u(a_2, w)) = u(a_1 \oplus a_2, w)
\]

Here, we assume that \( a_0 \) and \( \oplus \) are directly given for simplicity. In general, finding \( a_0 \) and \( \oplus \) for a given update program is an isolated synthesis task and thus can be treated separately.
Procedure. A segment tree is a tree-like data structure where each vertex corresponds to a segment in the list. On each vertex, several values with respect to the corresponding segment are maintained.

- For each update operation, the segment tree distributes the updated range into several disjoint vertices and applies the update in batch via lazy tags, which will be discussed later.
- For each query operation, the segment tree also distributes the updated range into disjoint vertices and merges the maintained values on these vertices together.

Figure 21 shows the sketch of segment trees. For simplicity, we assume the element in the list, the output of the query program $h$, and the parameter $a$ of the update program $u$ are all integers. This template uses an array $info$ to implement the segment tree, where

- $info[1]$ records the information on the root node, which corresponds to the whole list.
- $info[2k]$ and $info[2k+1]$ correspond to the left child and the right child of node $k$ respectively.
- For each node $k$, $info[k]$ records the output of $h$ and the auxiliary values on the segment corresponding to node $k$ (Lines 1-4).
- Array $tag$ records the lazy tag on each node. $tag[k]$ represents that all elements inside the range corresponding to node $k$ should be updated via $\lambda w. u \ (tag[k], w)$, but such an update has not been applied to those nodes strictly in the subtree of node $k$ yet.

There are several functions used in this template:

- apply deals with an update on all elements in the segment corresponding to node $pos$ by updating $info[pos]$ via combinator $comb_2$ (Line 7).
- pushdown applies the tag on node $pos$ to its children (Lines 11-12) and then clears it (Line 13).
- initialize initializes the information for node $pos$ which corresponds to range $[l, r]$. It recursively into two children (Lines 18-19) and merges the sub-results via $comb_1$ (Line 20).
- update applies an update $([ul, ur], \lambda w. u \ (a, w))$ to node $pos$ that corresponds to range $[l, r]$. If $[l, r]$ does not overlap with $[ul, ur]$, the update will be ignored (Line 23). If $[l, r]$ is contained by $[ul, ur]$, a lazy tag will be put (Line 24). Otherwise, update recurses into two children (Lines 26-27) and merges the sub-results via $comb_1$ (Line 28).
- query calculates a sub-result for query $[ul, ur]$ by considering elements in node $pos$ only. It is implemented similarly to update.

To solve a task, the segment tree is first initialized via function initialize (Line 37) and then responds to each operation by invoking the corresponding function (Lines 39-43).

Reduction to the lifting problem. To apply this paradigm, we need to find two combinators $comb_1$, $comb_2$ and an auxiliary program $aux$ satisfying the following condition, where $comb_1$ merges the sub-results on two sub-segments, $comb_2$ update the result after an update operation is applied to each element in the segment, and $aux$ provides necessary auxiliary values.

\[
\begin{align*}
(h \bowtie aux) \ (xs_L + xs_R) &= comb_1 \ ((h \bowtie aux) \ xs_L, (h \bowtie aux) \ xs_R) \\
(h \bowtie aux) \ (map (\lambda w. u \ (a, w)) \ xs) &= comb_2 \ (a, (h \bowtie aux) \ xs)
\end{align*}
\]

Similar to the second paradigm of the longest segment problem (Section 6.2), the two formulas above can be unified into a single lifting problem $LP(h, op)$, where $op$ is defined as below and its complementary input is from \{"merge", "update"\} × (the update set of $u$).

\[
\text{op} \ (\text{tag, a), } (xs_L, xs_R)) \triangleq \begin{cases} 
\text{xs}_L + \text{xs}_R & \text{tag = “merge”} \\
\text{map (\lambda w. u \ (a, w))} \ \text{xs}_L & \text{tag = “update”}
\end{cases}
\]

Time complexity. Let $n$ be the length of the initial list. Under the efficiency condition, one can verify that (1) the bottleneck of initialize is $O(n)$ invocations of $comb_1$, and (2) the bottleneck of query and update are $O(\log n)$ invocations of $comb_1$ and $comb_2$. Therefore, under the efficiency
struct Info {
  Int res; // Variable representing the output of h.
  // Variables representing the output of aux.
} info[N];
Int tag[N];

void apply(int pos, Int a){
  info[pos] = /*comb2(a, info[pos])*/;
  tag[pos] = tag[pos] ⊗ a;
}

void pushdown(int pos) {
  apply(pos * 2, tag[pos]);
  apply(pos * 2 + 1, tag[pos]);
  tag[pos] = a0;
}

void initialize(int pos, int *A, int l, int r) {
  if (l == r) {info[pos] = /*h[A[l]], aux[A[l]]*/; return;}
  int mid = l + r >> 1;
  initialize(pos * 2, A, l, mid);
  initialize(pos * 2 + 1, A, mid + 1, r);
  info[pos] = /*comb1(info[pos * 2], info[pos * 2 + 1])*/;
}

void update(int pos, int l, int r, int ul, int ur, Int a) {
  if (l > ur || r < ul) return;
  if (l >= ul && r <= ur) {apply(pos, a); return;}
  int mid = l + r >> 1; pushdown(pos);
  update(pos * 2, l, mid, ul, ur, a);
  update(pos * 2 + 1, mid + 1, r, ul, ur, a);
  info[pos] = /*comb1(info[pos * 2], info[pos * 2 + 1])*/;
}

Info query(int pos, int l, int r, int ul, int ur) {
  if (l > ur || r < ul) return {/*h[], aux[]*/};
  if (l >= ul && r <= ur) return info[pos];
  int mid = l + r >> 1; pushdown(pos);
  return /*comb1(query(pos * 2, 1, mid, ul, ur), query(pos * 2 + 1, mid +1, r, ul, ur))*/;
}

void range(int n, int *A, int m, Operator* op) {
  initialize(1, A, 0, n - 1);
  for (int i = 0; i < m; ++i) {
    if (op[i].type == Update) {
      update(1, 0, n - 1, op[i].l, op[i].r, op[i].a);
    } else {
      print(query(1, 0, n - 1, op[i].l, op[i].r));
    }
  }
}

Fig. 21. The template for the algorithmic paradigm of segment trees.

condition, the resulting program will take linear time to perform pre-processing and will respond to each operation in logarithmic time.