Decomposition-based Synthesis for Applying Divide-and-Conquer–like Algorithmic Paradigms

RUYI JI, Peking University, Beijing, China
YUWEI ZHAO, Peking University, Beijing, China
YINGFEI XIONG, Peking University, Beijing, China
DI WANG, Peking University, Beijing, China
LU ZHANG, Peking University, Beijing, China
ZHENJIANG HU, Peking University, Beijing, 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 article, 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 six different algorithmic paradigms. *AutoLifter* solves 82/96 tasks with an average time cost of 20.17 s, 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

ACM Reference Format:
Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024. Decomposition-based Synthesis for Applying Divide-and-Conquer–like Algorithmic Paradigms. *ACM Trans. Program. Lang. Syst.* 46, 2, Article 8 (June 2024), 59 pages. https://doi.org/10.1145/3648440

This work is supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902.

Authors' address: R. Ji, Y. Zhao, Y. Xiong (Corresponding author), D. Wang, L. Zhang, and Z. 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; e-mails: jiruyi910387714@pku.edu.cn, zhaoyuwei@stu.pku.edu.cn, xiongyf@pku.edu.cn, wangdi95@pku.edu.cn, zhanglucs@pku.edu.cn, huzj@pku.edu.cn.

ACM 0164-0925/2024/06-ART8
https://doi.org/10.1145/3648440

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 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 article 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 article, 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 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 subtasks 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 six 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 of 96 tasks with an average time cost of 20.01 s, 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 s.
- *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 article 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.

```
if len(xs) <= 1: return INF;
return sorted(xs)[1];
```

Fig. 1. Second minimum.

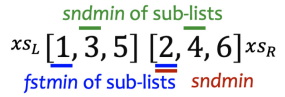


Fig. 2. An example of calculating *sndmin*.

$$
\begin{aligned}
&aux\ xs = min\ xs\\
&comb\ ((smin_L, aux_L), (smin_R, aux_R)) =\\
&\quad \textbf{let}\ csmin = \min(smin_L, smin_R, \max(aux_L, aux_R))\ \textbf{in}\\
&\quad\quad \textbf{let}\ caux = \min(aux_L, aux_R)\ \textbf{in}\\
&\quad\quad\quad (csmin, caux)
\end{aligned}
$$

Fig. 3. *aux* and *comb* for *sndmin*.

```
def dac(xs, l, r):
  if r - l <= 1:
    return (orig([xs[l]]), aux([xs[l]]))
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return comb(lres, rres)
return dac(xs, 0, len(xs))[0]
```

Fig. 4. A divide-and-conquer template on lists.

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 *sndmin* be a function returning the second-smallest value in an integer list. A natural implementation of *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 *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 $sndmin\ xs_L$ and $sndmin\ xs_R$, and then combines them into $sndmin\ xs$. In this procedure, a combinator $comb$ satisfying the formula below is required, where $xs_L +\!\!+ xs_R$ represents the concatenation of two lists,

$$sndmin\ (xs_L +\!\!+ xs_R) = comb\ (sndmin\ xs_L, sndmin\ xs_R).$$

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 *sndmin* with a program *aux* (denoted as an *auxiliary program*) specifying necessary auxiliary values to make a valid combinator *comb* exist, as follows:

$$
\begin{aligned}
sndmin'\ (xs_L +\!\!+ xs_R) &= comb\ (sndmin'\ xs_L, sndmin'\ xs_R)\\
\textbf{where}\ sndmin'\ xs &\triangleq (sndmin\ xs, aux\ xs).
\end{aligned}
\tag{1}
$$

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 *orig*

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	$N_{\mathbb{Z}} + N_{\mathbb{Z}} \mid min\ N_{\mathbb{L}}$
		$\mid$	$max\ N_{\mathbb{L}} \mid sum\ N_{\mathbb{L}}$
List expr	$N_{\mathbb{L}}$	$\rightarrow$	Input

(a) The program space $\mathcal{L}^{ex}_{aux}$ of *aux*.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	Inputs $\mid \min(N_{\mathbb{Z}}, N_{\mathbb{Z}})$
		$\mid$	$N_{\mathbb{Z}} + N_{\mathbb{Z}} \mid \max(N_{\mathbb{Z}}, N_{\mathbb{Z}})$

(b) The program space $\mathcal{L}^{ex}_{comb}$ of *comb*.

Fig. 5. A solution space for synthesizing a D&C program of *sndmin*, where the output of *aux* can be a tuple of integers, representing the usage of multiple auxiliary values (Figure 5(a)), and the output of *comb* can also be a tuple, since *comb* usually needs to calculate multiple values (Figure 5(b)).

stands for the original program *sndmin* (Figure 1). In this template, function *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 *aux* are applied directly. Otherwise, *dac* is recursively invoked on the two halves of the sub-list, and the results are combined by *comb*. Note that although *aux* is applied only to singleton lists in this template, it is defined for all lists to guide the design of the *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 **syntax-guided synthesis (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. However, 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.

```
fstmin, sndmin = INF, INF
for v in xs:
  sndmin = min(sndmin, max(fstmin, v))
  fstmin = min(fstmin, v)
return sndmin
```

Fig. 6. A single-pass program for *sndmin*.

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.

— Similarly 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?*

[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.

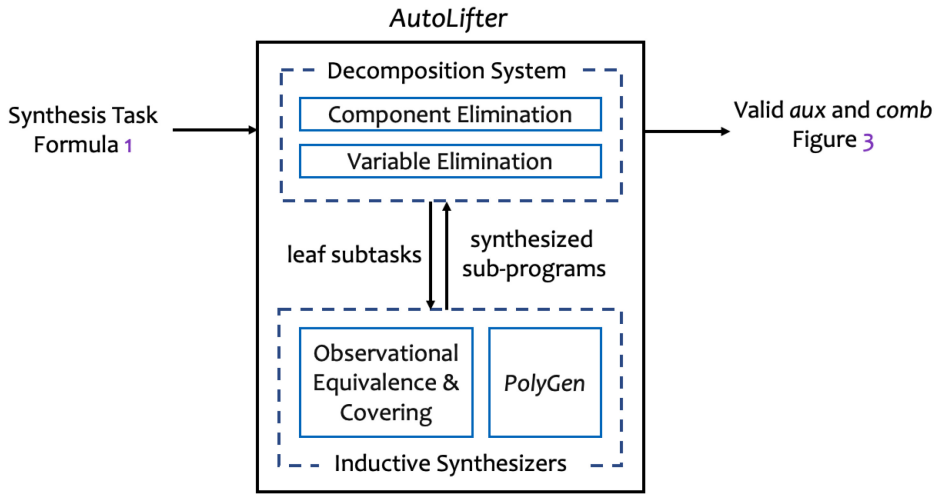


Fig. 7. The workflow of *AutoLifter*.

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.

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 construct *comb* as follows:

$$comb\ in \triangleq (comb_1\ in, comb_2\ in).$$

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,

$$sndmin\ (xs_L +\!\!+ xs_R) = comb_1\ (sndmin'\ xs_L, sndmin'\ xs_R), \textbf{where}\ sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs), \tag{2}$$

$$aux\ (xs_L +\!\!+ xs_R) = comb_2\ (sndmin'\ xs_L, sndmin'\ xs_R), \textbf{where}\ sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs). \tag{3}$$

Both specifications involve the same unknown program *aux*. If we synthesize from these specifications individually, then 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 *aux* provides more information than necessary. Consequently, these two requirements can be satisfied in order. We can first find *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 as follows:

— 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 *aux* as *min* and take $comb_1$ as the *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 aux' denotes the new auxiliary values needed to calculate auxiliary values themselves,

$$\begin{aligned} &aux\ (xs_L +\!\!+ xs_R) = comb_2\ (sndmin'\ xs_L, sndmin'\ xs_R) \\ &\textbf{where}\ sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs) \\ &\qquad\qquad aux\ xs \triangleq (min\ xs, aux'\ xs). \end{aligned} \tag{4}$$

In the *sndmin* example, no other information is needed for calculating the auxiliary value of the first minimum. One possible result here is to take aux' as an empty program (i.e., returns an empty tuple) and take $comb_2$ as the *caux* 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.

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, $comb_1$ and *aux*. Our second decomposition method *variable elimination* decomposes this task by deriving a subtask involving only *aux*. In other words, this method eliminates a program variable (i.e., $comb_1$) from the specification.

To derive a specification only for *aux*, we first revisit the fundamental reason why *aux* is needed. Let us consider two pairs of lists, (xs_L, xs_R) in Figure 2 and another pair (xs'_L, xs'_R),

$$xs_L\ [1,3,5], [2,4,6]\ xs_R \qquad xs'_L\ [1,3,5], [1,4,6]\ xs'_R. \tag{5}$$

Although the second minimums of xs_L and xs_R (3 and 4) are the same as their counterparts of (xs'_L, xs'_R), the second minimum of the combined list $xs_L +\!\!+ xs_R$ (which is 2) differs from that of $xs'_L +\!\!+ xs'_R$ (which is 1). Consequently, if *aux* is not involved, then a conflict will emerge after

substituting these two list pairs into the specification of $comb_1$ (Formula (2)). Specifically, (xs_L, xs_R) requires $comb_1$ to output 1 from input $(3, 4)$ but (xs'_L, xs'_R) requires $comb_1$ to output 2 from the same input. Such a $comb_1$ does not exist, because it must produce the same output from the same input.

The above analysis indicates that a necessary condition on *aux* is to ensure that a function exists for $comb_1$ to implement. When the expected outputs of $comb_1$ (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 *aux* and thus separates the synthesis of *aux* and $comb_1$. 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 $comb_1$ must be different when their respective outputs differ,

$$\begin{aligned} &sndmin\ (xs_L +\!\!+ xs_R) \neq sndmin\ (xs'_L +\!\!+ xs'_R) \\ &\qquad \rightarrow (sndmin'\ xs_L, sndmin'\ xs_R) \neq (sndmin'\ xs'_L, sndmin'\ xs'_R) \\ &\qquad\qquad \textbf{where}\ sndmin' xs \triangleq (sndmin\ xs, aux\ xs). \end{aligned}$$

For clarity, we transform this specification into the following equivalent form to make the constraint on *aux* clear. One possible result here takes *aux* as *min*,

$$\begin{aligned} &(sndmin\ xs_L, sndmin\ xs_R) = (sndmin\ xs'_L, sndmin\ xs'_R) \\ &\quad \wedge\ sndmin\ (xs_L +\!\!+ xs_R) \neq sndmin\ (xs'_L +\!\!+ xs'_R) \\ &\qquad \rightarrow (aux\ xs_L, aux\ xs_R) \neq (aux\ xs'_L, aux\ xs'_R). \end{aligned} \tag{6}$$

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 $comb_1$ that calculates the second minimum using the auxiliary value found in the first subtask,

$$sndmin\ (xs_L +\!\!+ xs_R) = comb_1\ \big((sndmin\ xs_L, min\ xs_L), (sndmin\ xs_R, min\ xs_R)\big). \tag{7}$$

One possible result here takes $comb_1$ as the *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 *aux* (represented by Formula (6)) and the other for sub-programs of *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 xs_L, xs_R, xs'_L, and xs'_R) 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 *comb* (e.g., Formula (7)) is already in the input–output form, where input–output examples are available. For example, under example $(xs_L, xs_R) \triangleq ([1, 3, 5], [2, 4, 6])$, Formula (7) requires sub-program $comb_1$ 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 *aux* (e.g., Formula (6)) is still relational. It involves the outputs of *aux* on four different inputs (xs_L, xs_R, xs'_L, xs'_R), making input–output examples unavailable. Fortunately, a domain property here is that the size of *aux* is usually much smaller than *comb*. Specifically, since *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 *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 **observational equivalence (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 $comb_1$ to implement but does not ensure that such a program exists in the program space ($\mathcal{L}^{ex}_{comb}$, Figure 5(b)). One can verify that $(min\ xs) + (min\ xs)$ is also valid for this subtask, but a corresponding combinator does not exist in $\mathcal{L}^{ex}_{comb}$.

There are two possible strategies for using such approximate decomposition methods in practice as follows:

- (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*

Table 1. Counter-examples of *sum xs* and *max xs* for Formula (6)

Program	(xs_L, xs_R)	(xs'_L, xs'_R)	Simplified Specification
sum xs	$([0,2],[0,1,2])$	$([0,2],[1,1,1])$	$(2,1) = (2,1) \wedge 0 \neq 1 \rightarrow (2,3) \neq (2,3)$
max xs	$([0,2],[0,2])$	$([0,2],[1,2])$	$(2,2) = (2,2) \wedge 0 \neq 1 \rightarrow (2,2) \neq (2,2)$

achieves this through a combined effect between the enumeration-based synthesizer (mainly OE) and the program space ($\mathcal{L}^{ex}_{aux}$, Figure 5(a)).

Programs in $\mathcal{L}^{ex}_{aux}$ 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 pinciple 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. Since *AutoLifter* synthesizes *aux* by enumerating programs from small to large, it prefers smaller programs and thus can successfully find *min xs* from those unnecessarily complex programs.

The second category includes the remaining programs not related to *min xs*. The specification for *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 *compressing* property. As a side effect of ensuring an efficient D&C program, program space $\mathcal{L}^{ex}_{aux}$ includes only programs mapping a list (whose size is unbounded) to a constant-sized tuple of integers. Such programs *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 $\mathcal{L}^{ex}_{aux}$ can hardly satisfy the specification (Formula (6)), because this specification requires *aux* to generate different outputs on a series of input pairs. For example, *sum xs* and *max xs* are two candidates in $\mathcal{L}^{ex}_{aux}$ that are not related to *min xs*. Both of them are rejected by Formula (6), and the corresponding counter-examples are listed in Table 1.

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

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

— First, we prove that the probability for *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 *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 *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 *lifting problems*, a novel class of synthesis tasks considered by *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

[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.

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 *aux* for specifying auxiliary values and a combinator *comb* for quickly updating the result, as follows:

$$sndmin'\ (append\ xs\ v) = comb\ (v, sndmin'\ xs), \textbf{where}\ sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs). \quad (8)$$

A valid solution to this specification is shown in Figure 8. Similarly 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.

```
aux xs = min xs
comb (v, (smin_pre, aux_pre)) =
  let csmin = min(smin_pre, max(aux_pre, v)) in
    let caux = min(aux_pre, v) in
      (csmin, caux)
```

Fig. 8. *aux* and *comb* for incrementalization.

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 *sndmin* (Section 2.1).

- In both tasks, a list is created from existing lists via an operator ($xs_L +\!\!+ xs_R$ and $append\ xs\ v$) and the output of an original program (*sndmin*) on the created list is calculated.
- Both tasks aim at finding (1) a program *aux* (denoted as an *auxiliary program*) for specifying auxiliary values, and (2) a corresponding combinator *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 article, 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 $a, xs, func$ 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, $func_1 \mathbin{\triangle} func_2$ to apply two functions to the same value, $func_1 \times func_2$ to apply two functions to the two components in a pair, and $func^n$ to apply a function to each component in an n-tuple,

$$(func_1 \mathbin{\triangle} func_2)\ x \triangleq (func_1\ x, func_2\ x) \qquad (func_1 \times func_2)\ (x_1, x_2) \triangleq (func_1\ x_1, func_2\ x_2)$$
$$func^n\ (x_1, \ldots, x_n) \triangleq (func\ x_1, \ldots, func\ x_n).$$

Problem definition. Given an original program *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 *orig* a can be calculated from *orig* $a_1, \ldots,$ *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 *orig*, whose input type is denoted as A.
- An operator *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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$, each specified by a grammar and the corresponding interpretations (i.e., semantics), defining the spaces of candidate programs.

Table 2. The Correspondence between the Lifting Problem and Previous Synthesis Tasks

Paradigm	Specification	*orig*	*A*	*op*	*n*	*C*
D&C	Formula (1)	*sndmin*	`List`	$op\ ((), (xs_L, xs_R)) \triangleq xs_L +\!\!+ xs_R$	2	`Unit`
incrementalization (list append)	Formula (8)			$op\ (c, (xs)) \triangleq append\ xs\ c$	1	`Int`

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

$$(orig \vartriangle aux)\ (op\ (c, \overline{a})) = comb\ (c, (orig \vartriangle aux)^n\ \overline{a}). \tag{9}$$

Following the notations in Section 2, we mark the known programs given in the synthesis task (e.g., original program *orig* and operator *op*) as blue, the unknown programs to be synthesized as red, and those universally quantified values as green. Furthermore, we shall also omit the range of a universally quantified value (such as $\forall\, \overline{a} \in A^n$ 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)).

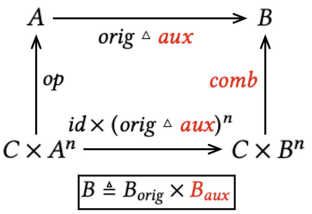
$$orig'\ \big(op\ (c, (a_1, \ldots, a_n))\big) = comb\ (c, (orig'\ a_1, \ldots, orig'\ a_n)), \textbf{where}\ orig'\ x \triangleq (orig\ x, aux\ x).$$

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 SyGuS problem [Alur et al. 2013], where the two languages $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ 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 $\mathcal{L}_{comb}$ includes only programs running in constant time.

In this article, we assume that $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ are implicitly given and denote a lifting problem as LP(*orig*, *op*). Besides, we assume that $\mathcal{L}_{aux}$ 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 $orig \vartriangle aux$. This diagram shows that $orig \vartriangle aux$ is a homomorphism mapping from *A* to *B*, where the operator *op* (related to *A*) is preserved as the combinator *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 *op* and immediately consumes it via *orig*; in contrast, the right-hand side avoids this construction by directly calculating from existing results, via the synthesized combinator *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,

```
mss = -INF
for i in range(len(x)):
  for j in range(i, len(x)):
    mss = max(mss, sum(x[i: j+1]))
return mss
```

Fig. 9. Maximum segment sum.

Fig. 10. An example of calculating *mss*.

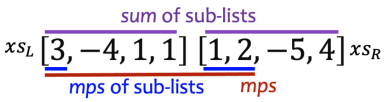


Fig. 11. An example of calculating *mps*.

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, similarly to the second minimum example, if we divide the input list into two halves, then 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], that of the right half is the prefix list [1, 2], 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.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	$N_{\mathbb{Z}} + N_{\mathbb{Z}} \mid min\ N_{\mathbb{L}}$
		$\mid$	$max\ N_{\mathbb{L}} \mid sum\ N_{\mathbb{L}}$
		$\mid$	$mps\ N_{\mathbb{L}} \mid mts\ N_{\mathbb{L}}$
List expr	$N_{\mathbb{L}}$	$\rightarrow$	Input

```
mps xs = max([sum(xs[:i+1])
              for i in range(lens(xs))])
mts xs = max([sum(xs[i:])
              for i in range(lens(xs))])
```

Fig. 12. The extended program $\mathcal{L}^{mss}_{aux}$ of *aux* for the *mss* example, where semantics of *mps* and *mts* are explained using a Python-like syntax.

$$\begin{aligned}
&\boxed{a@mps}\ \boxed{a@mts}\ \boxed{a@sum}\\
&aux\ xs = (mps\ xs,\ mts\ xs,\ sum\ xs)\\
&comb\ ((), (res_L, res_R)) =\\
&\quad \textbf{let}\ (mss_L, (mps_L, mts_L, sum_L)) = res_L\ \textbf{in}\\
&\quad \textbf{let}\ (mss_R, (mps_R, mts_R, sum_R)) = res_R\ \textbf{in}\\
&\boxed{c@mss}\ \textbf{let}\ cmss = \max(mss_L, mss_R, mts_L + mps_R)\ \textbf{in}\\
&\boxed{c@mps}\ \textbf{let}\ cmps = \max(mps_L, sum_L + mps_R)\ \textbf{in}\\
&\boxed{c@mts}\ \textbf{let}\ cmts = \max(mts_L + sum_R, mts_R)\ \textbf{in}\\
&\boxed{c@sum}\ \textbf{let}\ csum = sum_L + sum_R\ \textbf{in}\\
&\qquad (cmss, (cmps, cmts, csum))
\end{aligned}$$

Fig. 13. The expected *aux* and *comb* for *mss*.

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.

The task of applying D&C to *mss* can be regarded as a lifting problem LP(*mss*, *op*) for operator $op\ ((), (xs_L, xs_R)) \triangleq xs_L +\!\!+ xs_R$. The raw specification of this lifting problem is as follows:

$$(mss \triangle aux)\ (xs_L +\!\!+ xs_R) = comb\ ((), (mss \triangle aux)^2\ (xs_L, xs_R)),$$

which can be transformed into a more readable form as follows:

$$mss'\ (xs_L +\!\!+ xs_R) = comb\ \big((), (mss'\ xs_L, mss'\ xs_R)\big), \textbf{where}\ mss' \triangleq mss \triangle aux. \tag{10}$$

For simplicity, in this example, we continue using $\mathcal{L}^{ex}_{comb}$ (Figure 5(b)) and extend $\mathcal{L}^{ex}_{aux}$ (Figure 5(a)) by directly introducing *mps* and *mts* as language constructs ($\mathcal{L}^{mss}_{aux}$, Figure 12). Note that the full languages used in our implementation are formed by more primitive constructs where, for example, *mps* is implemented as $max\ (scanl\ (+)\ xs)$ (Example 7.1, Section 7).

Figure 13 shows the expected solution synthesized from $\mathcal{L}^{mss}_{aux}$ and $\mathcal{L}^{ex}_{comb}$, where *aux* returns a 3-tuple and *comb* returns a 4-tuple. This solution is formed by expressions for producing components in the output tuples. We label these expressions in Figure 13 for later reference.

4.2 Decomposition System

AutoLifter decomposes lifting problems using two decomposition methods, *component elimination* and *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 *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.

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 as follows:

$$\forall (x, y) \in S, (out\ x, g\ x) = f\ (in\ y, g\ y). \tag{11}$$

In this specification, *out* and *in* are known functions with the same input type, S is a set of input pairs, each comprising an input of *out* and an input of *in*, and f and g are two unknown programs

to be synthesized. Intuitively, g specifies some auxiliary values, and f calculates the output of *out* and the new auxiliary values from the output of *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 $f \triangleq f_1 \vartriangle f_2$, 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, then we can assume g is formed by two sub-programs g_1 and g_2 such that $g \triangleq g_1 \vartriangle g_2$, 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,

$$\forall (x, y) \in S, out\ x = f_1\ (in\ y, g_1\ y). \tag{12}$$

(2) If g_1 is *null*, i.e., no auxiliary value is needed, then a solution for the full specification (Formula (11)) is $f \triangleq f_1 \vartriangle null$ and $g \triangleq null$. 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,

$$\forall (x, y) \in S, (g_1\ x, g_2\ x) = f_2\ ((in \vartriangle g_1)\ y, g_2\ y). \tag{13}$$

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,

$$f\ (v_{in}, (v_1, v_2)) \triangleq \left(f_1\ (v_{in}, v_1), f_2\ ((v_{in}, v_1), v_2)\right) \qquad g\ y \triangleq (g_1\ y, g_2\ y).$$

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,

$$(orig\ (op\ (c, \overline{a})), aux\ (op\ (c, \overline{a}))) = (comb\ (c\ (orig^n\ \overline{a}, aux^n\ \overline{a})). \tag{14}$$

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:

General form: $(out\ x, g\ x) = f\ (in\ y, g\ y)$.

General Form	Lifting Problem	General From	Current Task
f	$comb$	in	$orig^n$
g	aux	out	$orig$
x	$op\ (c, \overline{a})$		
y	$\overline{a}$		

The two differences here are that (1) *comb* has an extra parameter c and (2) *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 *comb* as a pair of two sub-programs $comb_1$ and $comb_2$, assume the form of *aux* as a pair of two sub-programs aux_1 and aux_2, and then decompose the lifting problems as follows.

(1) The first subtask synthesizes $comb_1$ and aux_1 from the following specification:

$$orig\ (op\ (c, \overline{a})) = comb_1\ \big(c, (orig^n\ \overline{a}, {aux_1}^n\ \overline{a})\big). \tag{15}$$

(2) If the synthesis result of aux_1 is *null*, which means no auxiliary value is needed to calculate the expected output of *orig*, then a solution for the lifting problem is $comb \triangleq comb_1 \vartriangle null$ and $aux \triangleq null$. At this time, the second subtask is not needed.

(3) Otherwise, the second subtask synthesizes $comb_2$ and aux_2 from the specification below, where $comb_1$ and aux_1 denote the result of the first subtask. This task can be recursively solved in the same way,

$$(aux_1 \vartriangle aux_2)\ (op\ (c, \overline{a})) = comb_2\ \big(c, ((orig \vartriangle aux_1)^n\ \overline{a}, {aux_2}^n\ \overline{a})\big). \tag{16}$$

(4) Given the results of the two subtasks, a solution for the lifting problem can be obtained by taking *aux* as $aux_1 \vartriangle aux_2$ and constructing *comb* by pairing up $comb_1$ and $comb_2$ with properly adjusting the structure of their inputs.

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

$$mss\ (xs_L +\!\!+ xs_R) = comb_1\ \big((), ((mss\ xs_L, mss\ xs_R), (aux_1\ xs_L, aux_1\ xs_R))\big). \tag{17}$$

One can verify that components $a@mps$, $a@mts$ and $c@mss$ in Figure 13 form a valid solution for this subtask. Given this solution, the specification of the second subtask is as follows:

$$(aux_1 \vartriangle aux_2)\ (xs_L +\!\!+ xs_R) = comb_2\ \big((), ((known\ xs_L, known\ xs_R), (aux_2\ xs_L, aux_2\ xs_R))\big)$$
$$\textbf{where}\ aux_1 \triangleq (mps \vartriangle mts)\ \text{and}\ known \triangleq mss \vartriangle aux_1.$$

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

$$aux_1\ (xs_L +\!\!+ xs_R) = comb_{2.1}\ \big((), ((known\ xs_L, known\ xs_R), (aux_{2.1}\ xs_L, aux_{2.1}\ xs_R))\big)$$
$$\textbf{where}\ aux_1 \triangleq (mps \vartriangle mts)\ \text{and}\ known \triangleq mss \vartriangle aux_1.$$

One can verify that components $a@sum$, $c@mps$, and $c@mts$ in Figure 13 form a valid solution for subtask 2.1, which leads to the following subtask 2.2:

$$(aux_{2.1} \vartriangle aux_{2.2})\ (xs_L +\!\!+ xs_R) = comb_{2.2}\ \big((), ((known\ xs_L, known\ xs_R), (aux_{2.2}\ xs_L, aux_{2.2}\ xs_R))\big)$$
$$\textbf{where}\ aux_{2.1} \triangleq sum\ \text{and}\ known \triangleq \big(mss \vartriangle (mps \vartriangle mts)\big) \vartriangle aux_{2.1}.$$

This subtask can still be recursively decomposed by component elimination. At this time, we shall get *null* for the first sub-program of $aux_{2.2}$, 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 aux_1 and $comb_1$ 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,

$$\forall (x, y) \in S, x = f\ (in\ g\ y). \tag{18}$$

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 *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 following table:

$$\begin{array}{ll} \text{General form :} & x = f\ (in\ g\ y) \\ \text{Current task :} & orig\ (op\ (c, \overline{a})) = comb_1\ \left(c, (orig^n\ \overline{a}, {aux_1}^n\ \overline{a})\right) \end{array}$$

General Form	Current Task
f	$comb_1$
g	aux_1
x	$orig\ (op\ (c, \overline{a}))$
y	$(c, \overline{a})$
in	$\lambda aux_1.\ \lambda (c, \overline{a}).\ \left(c, (orig^n\ \overline{a}, {aux_1}^n\ \overline{a})\right)$

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 following specification:

$$orig\ (op\ (c, \overline{a})) \neq orig\ (op\ (c', \overline{a'})) \rightarrow \left(c, (orig^n\ \overline{a}, {aux_1}^n\ \overline{a})\right) \neq \left(c', (orig^n\ \overline{a'}, {aux_1}^n\ \overline{a'})\right).$$

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,

$$\left(orig^n\ \overline{a} = orig^n\ \overline{a'} \wedge orig\ (op\ (c, \overline{a})) \neq orig\ (op\ (c, \overline{a'}))\right) \rightarrow {aux_1}^n\ \overline{a'} \neq {aux_1}^n\ \overline{a'}. \tag{21}$$

(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,

$$orig\ (op\ (c, \overline{a})) = comb_1\ \left(c, (orig^n\ \overline{a}, {aux_1}^n\ \overline{a})\right). \tag{22}$$

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 as follows:

$$\begin{aligned} (mss\ xs_L, mss\ xs_R) = (mss\ xs'_L, mss\ xs'_R) \wedge mss\ (xs_L +\!\!+ xs_R) \neq mss\ (xs'_L +\!\!+ xs'_R) \\ \rightarrow (aux_1\ xs_L, aux_1\ xs_R) \neq (aux_1\ xs'_L, aux_1\ xs'_R). \end{aligned}$$

One can verify that components $a@mps$ and $a@mts$ form a valid solution to this subtask.

ALGORITHM 1: A greedy implementation of the decomposition system.

Input: A lifting problem LP(*orig*, *op*).
Output: A solution (*aux*, *comb*) to lifting problem LP(*orig*, *op*).

```
1  Function VariableElimination(a subtask in the form of Formula (15)):
2      subtask_1 ← the first subtask generated by variable elimination (Formula (21));
3      aux_1 ← InductiveSynthesisForAux(subtask_1);
4      subtask_2 ← the second subtask corresponding to aux_1 (Formula (22));
5      return (aux_1, InductiveSynthesisForComb(subtask_2));
6  Function ComponentElimination)a lifting problem in the form of Formula (14):
7      subtask_1 ← the first subtask generated by component elimination (Formula (15));
8      (aux_1, comb_1) ← VariableElimination(subtask_1);
9      if aux_1 = null then return (null, comb_1 ▵ null);
10     subtask_2 ← the second subtask corresponding to (aux_1, comb_1) (Formula (16));
11     (aux_2, comb_2) ← ComponentElimination(subtask_2);
12     return (aux, comb) constructed from (aux_1, comb_1) and (aux_2, comb_2);
13 return ComponentElimination (the original lifting problem);
```

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 and 8), and the second one is solved by recursively applying component elimination (Lines 10 and 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 two components) is greatly reduced compared to the original lifting problem (#1, including seven 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.

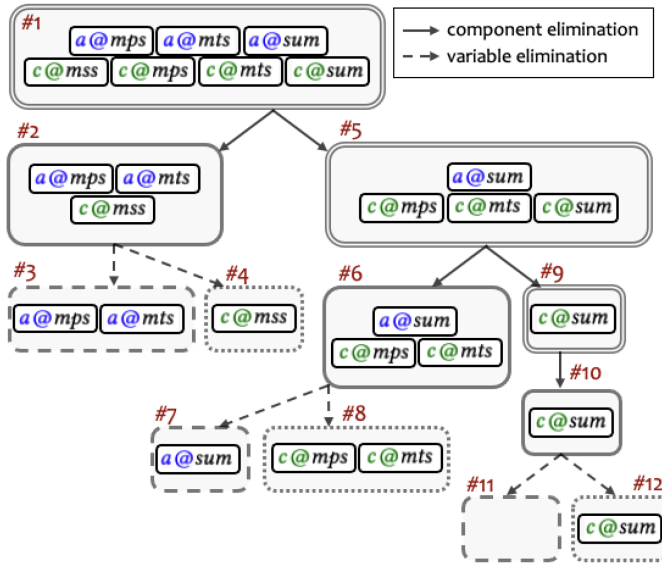


Fig. 14. The decomposition performed by the decomposition system to synthesize programs in Figure 13.

ALGORITHM 2: CEGIS framework

Input: A specification $\Phi = \forall \overline{x}, \phi(prog, \overline{x})$.
Output: A valid program.

1 $examples \leftarrow \emptyset$;
2 **while** *true* **do**
3 $\quad prog \leftarrow$ `Synthesis`$(\forall \overline{x} \in examples, \phi(prog, \overline{x}))$;
4 $\quad e \leftarrow$ `CounterExample`$(prog, \Phi)$;
5 $\quad$ **if** $e = \bot$ **then return** $prog$;
6 $\quad examples \leftarrow examples \cup \{e\}$;
7 **end**

ALGORITHM 3: Example-based solver of *comb*.

Input: An example set *examples* and programs $(orig, op, aux_1)$ specifying the task.
Output: A valid combinator $comb_1^*$.

1 $ioexamples \leftarrow \emptyset$;
2 **foreach** $(c, \overline{a}) \in examples$ **do**
3 $\quad input \leftarrow (c, (orig \triangleleft aux_1)^n\ \overline{a})$;
4 $\quad output \leftarrow orig\ (op\ (c, \overline{a}))$;
5 $\quad ioexamples \leftarrow ioexamples \cup \{(input, output)\}$;
6 **end**
7 **return** `SynthesisFromIOExamples`$(ioexamples)$;

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 *prog* from existing examples (Line 3) and the verifier generates a counter-example e under which *prog* is incorrect, i.e., $\neg\phi(prog, e)$ 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 article, 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

Table 3. Two Possible Examples Generated from the Subtask in Example 4.2

Id	(xs_L, xs_R)	(xs'_L, xs'_R)	premise	requirement
$\overline{xs}_1$	$([1],[1])$	$([1],[-1,1])$	$(1,1) = (1,1) \wedge 2 \neq 1$	$(aux_1\ [1], aux_1\ [1]) \neq (aux_1\ [1], aux_1\ [-1,1])$
$\overline{xs}_2$	$([1],[1])$	$([1,-1],[1])$	$(1,1) = (1,1) \wedge 2 \neq 1$	$(aux_1\ [1], aux_1\ [1]) \neq (aux_1\ [1,-1], aux_1\ [1])$

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 *comb*. We begin with the leaf subtask of *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 $comb_1$ to output $orig\,(op\,(c, \overline{a}))$ from input $(c, (orig \triangleleft aux_1)^n\ \overline{a})$. 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 *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 *aux*. For the leaf subtask of *aux* (Formula (21)), an example is an assignment to $(c, \overline{a}, \overline{a'})$, with a constraint of ${aux_1}^n\ \overline{a} \neq {aux_1}^n\ \overline{a'}$. 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.

Example 4.4. Table 3 shows two examples possibly generated from the subtask in Example 4.2. The maximum prefix sum *mps* satisfies example $\overline{xs}_1$, 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 $\overline{xs}_2$, and their pair $mts \triangleleft mps$ satisfies both examples.

The example-based synthesizer for *aux* (Algorithm 4) is built upon an existing enumerative synthesizer, namely *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 and 2, Algorithm 4).

Example 4.6. When the example set is $\{\overline{xs}_1, \overline{xs}_2\}$ (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,

ALGORITHM 4: Example-based solver of *aux*.

Input: An example set *examples* and an integer lim_c specifying the number of components considered by observational covering.
Output: A valid auxiliary program aux_1.

```
1  involvedInputs ← {a | (c, ā, ā′) ∈ examples ∧ (a ∈ ā ∨ a ∈ ā′)};
2  oe ← ObservationalEquivalenceSolver(involvedInputs);
3  ∀size ≥ 0, programs[size] ← [];  result ← ⊥;
4  Function IsCovered(prog, size):
5  |  return ∃size' ≤ size, ∃prog′ ∈ programs[size′], prog′ satisfies all examples that are satisfied by prog;
6  Function Insert(prog, size):
7  |  if prog satisfies all examples ∧result = ⊥ then result ← prog;
8  |  if ¬IsCovered(prog, size) then programs[size].Append(prog);
9  Function Extend():
10 |  component ← oe.Next(); Insert(component, 1)
11 |  prePrograms ← programs;
12 |  foreach size ∈ [1, ..., lim_c − 1] and prog ∈ prePrograms[size] do
13 |  |  Insert(prog △ component, size + 1);
14 |  end
15 Insert(null, 0);
16 while result = ⊥ do Extend();
17 return result;
```

mps and *mts* are both required for calculating *mss* in D&C. In these cases, the form of aux_1 can be assumed as a tuple of components (i.e., $comp_1 \triangle \ldots \triangle comp_k$), 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 *aux* maintains a program storage *programs* during the enumeration, where *programs*[*size*] stores existing programs formed by *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 lim_c and considers only combining at most 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 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 *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 $(c, \overline{a}, \overline{a'})$ 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 lim_c components, the effect of a program *prog* is covered by another program *prog*′ if (1) *prog*′ uses fewer components than *prog* and (2) *prog*′ satisfies all examples satisfied by *prog*. At this time, the covered program *prog* can be safely skipped from the top-level combination.

Example 4.7. When the example set is $\{\overline{xs}_1, \overline{xs}_2\}$ (Example 4.4), the effect of *max* is covered by *null* (the empty auxiliary program), because both programs satisfy no example and *max* uses one more component. Therefore, *max* can be safely skipped from the combination: Whenever there

is a program combined from *max* (assumed as *prog* $\triangle$ *max*) satisfying both examples, there must exist another valid program *prog* $\triangle$ *null* (i.e., *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 $\{\overline{xs}_1, \overline{xs}_2\}$ (Example 4.4), the limit lim_c is 2, and the first three components returned by OE are *max*, *mps*, and *mts* in order.

- Before the first invocation of `Extend`, *null* is inserted and the storage is $programs[0] = [null]$.
- In the first invocation, OE generates *max* and no other program is constructed. *max* will not be inserted into the storage as it is covered by *null*. So the storage will remain unchanged.
- In the second invocation, OE generates *mps* and no other program is constructed. *mps* is not covered by *null* as it satisfies $\overline{xs}_1$, an example violated by *null*. So *mps* will be inserted, and the storage will become $programs[0] = [null]$, $programs[1] = [mps]$.
- In the third invocation, OE generates *mts* and program *mps* $\triangle$ *mts* is generated by combination. Both programs are not covered and will be inserted, and the storage will become as follows:

$$programs[0] = [null] \quad programs[1] = [mps, mts] \quad programs[2] = [mps \triangle mts].$$

Then *mps* $\triangle$ *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 article are available in Appendix A.1. □

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 *aux* (Algorithm 4). Specifically, each subtask generated by *AutoLifter* depends only on the original lifting problem and the sub-programs of *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 *sndmin* 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 $\mathcal{L}_{aux}$ 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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ 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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ 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 ($\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$) 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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ 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 $\mathsf{LP}(orig, op)$, we construct a corresponding probabilistic model $\mathcal{M}[orig, op]$ by modeling the semantics of programs in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ as uniformly random functions. The detailed construction of $\mathcal{M}[orig, op]$ is as follows:

— $\mathcal{M}[orig, op]$ is constructed on a set of given parameters, including (1) the types and semantics of *orig* and *op* and (2) the syntax and the type of every program in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$. The only thing this model does is to assign random semantics to programs in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$.
— For simplicity, we make the following assumptions when constructing $\mathcal{M}[orig, op]$:
(1) Programs in $\mathcal{L}_{aux}$ are formed as tuples of components (i.e., $comp_1 \mathbin{\triangle} \ldots \mathbin{\triangle} comp_k$), where each component $comp_i$ outputs only a single auxiliary value.
(2) There is a universal value type V capturing the types of the output of *orig*, every auxiliary value, and the complementary input required by *op*.
(3) The numbers of different values in the input type A (the input type of *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 $s_A \gg s_V$.
— $\mathcal{M}[orig, op]$ generates a lifting problem by independently and uniformly sampling the semantics for every candidate program in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$. For example, suppose that there is a program in $\mathcal{L}_{aux}$ with type $A \to V$. The semantics of this program will be uniformly drawn from functions mapping from A to V ($s_V^{s_A}$ possibilities in total).

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

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

$$\Pr_{\varphi \sim \mathcal{M}[orig,op]} \Big[\textit{AutoLifter} \text{ generates an unrealizable subtask from } \varphi \;\Big|\; \exists (aux, comb), \big(size(aux, comb) \le lim_s \wedge (aux, comb) \text{ is valid for } \varphi\big) \Big],$$

where $\varphi \sim \mathcal{M}[orig, op]$ represents that φ is a random lifting problem sampled from $\mathcal{M}[orig, op]$ and *size*(*aux*, *comb*) represents the total size of programs *aux* and *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 LP(*orig*, *op*) (Definition 5.3). This factor counts the number of independent examples that *aux* must satisfy to ensure a function exists for calculating the output of *orig*.

Definition 5.3 (Mismatch Factor). Given lifting problem LP(*orig*, *op*) and integer t, the *mismatch factor* of this lifting problem is at least t if there exists t pairs of values $(\overline{a_i}, \overline{a_i'}) \in A^n \times A^n$ satisfying the following two conditions.

— For each pair $(\overline{a_i}, \overline{a_i'})$, the inputs of *comb* when no auxiliary value is used (i.e., the output of $orig^n$) are the same but the outputs of *comb* (i.e., the output of $orig \circ op$) are different. In other words, every pair $(\overline{a_i}, \overline{a_i'})$ must satisfy the following formula:

$$\exists c \in C, \big(orig^n\ \overline{a_i} = orig^n\ \overline{a_i'} \wedge orig\ (op\ (c, \overline{a_i})) \neq orig\ (op\ (c, \overline{a_i'}))\big). \tag{23}$$

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

Recall that A denotes the input type of *orig*, n denotes the arity of *op*, and C denotes the type of the complementary input required by *op*.

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

$$\big(orig^n\ \overline{a} = orig^n\ \overline{a'} \wedge orig\ (op\ (c, \overline{a})) \neq orig\ (op\ (c, \overline{a'}))\big) \rightarrow aux_1{}^n\ \overline{a} \neq aux_1{}^n\ \overline{a'}.$$

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*, aux_1 (a sub-program of *aux*) needs to output differently on t independent pairs of inputs. Intuitively, with a larger mismatch factor, an incorrect aux_1 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(*orig*, *op*) *of which the mismatch factor is at least* t*, the size-limited unrealizable rate of AutoLifter is bounded as follows:*

$$unreal(orig, op, lim_s) \leq 2^w \exp(-t/s_V^{n \cdot w}), \textbf{\textit{where}}\ w \triangleq lim_c \cdot lim_s.$$

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 *orig* and auxiliary programs in $\mathcal{L}_{aux}$ 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 *orig* and *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 *sndmin* and the auxiliary program *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 ∞ when the length of lists tends to ∞ 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 $orig^n$ (whose output domain is V^n) to output the same on $\overline{a}$ and $\overline{a'}$ and requires $orig \circ op$ (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, that there is one pair $(\overline{a}, \overline{a'})$ satisfying this condition in every $O(s_V^n)$ pairs.[4] Since there are up to $S_A/(2n)$ 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* ϵ *tends to* 0 *when* $s_A/s_V^{w'}$ *tends to* ∞*, 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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ 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 $s_V > 1$, which 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., (*min xs*, *max xs*)) or performing invertible arithmetic operations (e.g., (*min xs*)+(*min xs*)). Many of them may lead to unrealizable subtasks, for example, the combination function corresponding to (*min xs*) + (*min xs*) cannot be implemented in $\mathcal{L}^{ex}_{comb}$ (Figure 5(b)).

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 $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$, 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 article, we consider a specific condition (denoted as the *efficiency condition*) for the domain-specific languages. This condition requires that (1) every program in $\mathcal{L}_{aux}$ runs in constant time on a constant-sized input and (2) every program in $\mathcal{L}_{comb}$ runs in constant time.

Example 6.1. The domain-specific languages $\mathcal{L}^{ex}_{aux}$ and $\mathcal{L}^{ex}_{comb}$ (Figure 5(a) and (b), 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 follows:

- 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.

```
res = (orig([]), aux([]))
for v in xs:
  res = comb(v, res)
return res[0]
```

Fig. 15. A template of single-pass.

```
info = (p([]), aux([]))
res, l = 0, 0
for r in range(len(xs)):
  info = comb1(xs[r], info)
  while l <= r and not info[0]:
    info = comb2(xs[l], info)
    l += 1
  res = max(res, r - l + 1)
return res
```

Fig. 16. A template of sliding window.

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 *orig*, an auxiliary program *aux* and a combinator *comb* satisfying the following formula are required:

$$orig'\ (xs +\!\!+ [v]) = comb\ (v, orig'\ xs), \textbf{where}\ orig' \triangleq orig \vartriangle aux.$$

This task is equivalent to that for incrementalization (Section 4.1) and can be regarded as a lifting problem $\mathsf{LP}(orig, op)$ with $op\ (v, (xs)) \triangleq xs +\!\!+ [v]$. 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 *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 $change : C \times A \mapsto A$ applying a change to an A-element. To incrementally update the result of the original program *orig* after each change, an auxiliary program *aux* and a combinator *comb* satisfying the following formula are required:

$$orig'\ (change\ (c, a)) = comb\ (c, orig'\ a), \textbf{where}\ orig' \triangleq orig \vartriangle aux.$$

This task can be regarded as $\mathsf{LP}(orig, op)$ with $op\ (c, (a)) \triangleq change\ (c, a)$. Under the efficiency condition, the resulting incremental program must run in $O(1)$ time per change, because only *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 $\{\text{“front”}, \text{“back”}\} \times \texttt{Int}$ and the corresponding *change* is as follows:

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

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 lth to the rth element in the input list 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 combinators $comb_1$ 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,

$$(p \triangle aux)\ (xs +\!\!+ [v]) = comb_1\ (v, (p \triangle aux)\ xs)$$
$$head\ xs = v \rightarrow (p \triangle aux)\ (tail\ xs) = comb_2\ (v, (p \triangle aux)\ xs).$$

The condition of $head\ xs = v$ is involved to allow the second combinator $comb_2$ 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 as follows, where the complementary input is in $\{\text{“}append\text{”}, \text{“}remove\text{”}\} \times \texttt{Int}$:

$$op\ ((tag, v), (xs)) \triangleq \begin{cases} xs +\!\!+ [v] & tag = \text{“}append\text{”} \\ tail\ xs & tag = \text{“}remove\text{”} \wedge head\ xs = v \\ [] & otherwise \end{cases}.$$

Under the efficiency condition, the resulting program runs in $O(n)$ time on a list of length n, because both aux_1 and 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 *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 *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 *aux* and two combinators $comb_1$ and $comb_2$ such that $(aux, comb_1)$ is a solution to the lifting problem of D&C and $(aux, comb_2)$ is a solution to the lifting problem of incrementalization; in other words, the formulas below need to be satisfied,

$$(orig \triangle aux)\ (xs_L +\!\!+ xs_R) = comb_1\ ((orig \triangle aux)\ xs_L, (orig \triangle aux)\ xs_R)$$
$$(orig \triangle aux)\ (change\ (c, xs)) = comb_2\ (c, (orig \triangle aux)\ xs).$$

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 *change* or a token *“d&c”* never used before,

$$op\ (c, (xs_L, xs_R)) \triangleq \textbf{if}\ c = \text{“}d\&c\text{”}\ \textbf{then}\ xs_L +\!\!+ xs_R\ \textbf{else}\ change\ (c, xs_L).$$

Let n be the length of the initial list. A segment tree invokes $comb_1$ and $comb_2$ $O(\log n)$ times when processing each operation (either a query or a change). Therefore, the resulting program must be $O(\log n)$ time per operation under the efficiency condition. More details on this guarantee can be found in Appendix B.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	IntConst $\mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid sum\ N_{\mathbb{L}} \mid len\ N_{\mathbb{L}} \mid head\ N_{\mathbb{L}}$
		$\mid$	$last\ N_{\mathbb{L}} \mid access\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid count\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid min\ N_{\mathbb{L}}$
		$\mid$	$max\ N_{\mathbb{L}} \mid neg\ N_{\mathbb{Z}}$
List expr	$N_{\mathbb{L}}$	$\rightarrow$	Input $\mid take\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid drop\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid rev\ N_{\mathbb{L}}$
		$\mid$	$map\ F_{\mathbb{Z}}\ N_{\mathbb{L}} \mid filter\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid zip \oplus N_{\mathbb{L}}\ N_{\mathbb{L}} \mid sort\ N_{\mathbb{L}}$
		$\mid$	$scanl \oplus N_{\mathbb{L}} \mid scanr \oplus N_{\mathbb{L}}$
Binary Operator	$\oplus$	$\rightarrow$	$+ \mid - \mid \times \mid min \mid max$
Integer Function	$F_{\mathbb{Z}}$	$\rightarrow$	$(+\ \text{IntConst}) \mid (-\ \text{IntConst}) \mid neg$
Boolean Function	$F_{\mathbb{B}}$	$\rightarrow$	$(< 0) \mid (> 0) \mid odd \mid even$

Fig. 17. The grammar of $\mathcal{L}_{aux}$.

7 IMPLEMENTATION

Our implementation of *AutoLifter* focuses on lifting problems related to integer lists and integers. It can be generalized to other cases if the corresponding types, operators, and grammars are provided.

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

Language $\mathcal{L}_{aux}$ 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 *scanl* in our $\mathcal{L}_{aux}$ 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 follows:

$$scanl\,(\oplus)\,[xs_1, \ldots, xs_n] \triangleq [xs_1, xs_1 \oplus xs_2, \ldots, xs_1 \oplus \cdots \oplus xs_n].$$

Using this operator, the maximum prefix sum of list xs can be implemented as $max\,(scanl\,(+)\,xs)$.

Language $\mathcal{L}_{comb}$ 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. Similarly to the case of $\mathcal{L}_{aux}$, we add operators for accessing and constructing tuples (i.e., (S, S) and $N_{\mathbb{T}}.i$) 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]

[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.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	IntConst $\mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid \mathit{if}\ N_{\mathbb{B}}\ \mathit{then}\ N_{\mathbb{Z}}\ \mathit{else}\ N_{\mathbb{Z}} \mid N_{\mathbb{T}}.i$
Bool expr	$N_{\mathbb{B}}$	$\rightarrow$	$\neg N_{\mathbb{B}} \mid N_{\mathbb{B}} \wedge N_{\mathbb{B}} \mid N_{\mathbb{B}} \vee N_{\mathbb{B}} \mid N_{\mathbb{Z}} \leq N_{\mathbb{Z}} \mid N_{\mathbb{Z}} = N_{\mathbb{Z}}$
Tuple expr	$N_{\mathbb{T}}$	$\rightarrow$	Input $\mid N_{\mathbb{T}}.i$
Binary Operator	$\oplus$	$\rightarrow$	$+ \mid - \mid \times \mid \mathit{div}$

Fig. 18. The grammar of $\mathcal{L}_{comb}$.

— 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?
— **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 $\mathcal{L}_{comb}$ 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 and 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.

Table 4. The Profile of Synthesis Tasks Considered in Our Evaluation

Problem	D&C	Single-pass	Longest Segment	Segment Tree	Total
#Task	36	39	8	13	96

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 following four algorithmic problems:

— *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 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 four of five 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 constructed.[7] 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 three, one, and four 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 3.2-GHz 6-Core Processor. Every execution is under a time limit of 300 s 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 the following three different aspects when comparing these solvers:

[6] The 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.
[7] A duplicated task involved by both *DPASyn* and our D&C dataset is ignored in this construction.
[8] A common alias of segment trees.

Table 5. The Results of Comparing *AutoLifter* with *Enum* and *Relish*

Solver	D&C				Single-pass			
	#Solved	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$	#Solved	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$
AutoLifter	**29**/36	20.01		×1.000	**33**/39	8.861		×**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	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$	#Solved	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$
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	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$				
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 effectiveness of synthesis, measured by the number of solved tasks.
— The efficiency of synthesis, measured by the average time cost of successful synthesis.
— 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 $\text{Time}_{\texttt{Base}}$ and $\text{Time}_{\texttt{Ours}}$ 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 $\text{Time}_{\texttt{Res}}$ 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 $t_{\texttt{Base}}/t_{\texttt{Ours}}$, where $t_{\texttt{Base}}$ and $t_{\texttt{Ours}}$ 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 $\text{Time}_{\texttt{Res}}$. 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 follows:

— 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*.[9]
— 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 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 as follows:

— 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 *Parsynt*.[10] Note

[9] A gold medal winner in international programming competitions helped us to verify the synthesized programs.
[10] For those tasks taken from *Parsynt*, we use the program in its original evaluation and fix the two bugs we found.

Table 6. The Results of Comparing *AutoLifter* with *Parsynt*

Solver	$\#\text{Solved}_{\texttt{Base}}$	$\#\text{Solved}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$	$\#\text{Aux}_{\texttt{SP}}$
Parsynt17	**19**/20	**19**/20	15.59	**8.552**	N/A	58.62%
Parsynt21	24/36	**29**/36	**6.856**	7.315	×1.201	40.54%

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].

— Similarly 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 $\#\text{Solved}_{\texttt{Base}}$ and $\#\text{Solved}_{\texttt{Ours}}$ report the number of tasks solved by each version of *Parsynt* and *AutoLifter*, respectively.

— Columns $\#\text{Time}_{\texttt{Base}}$ and $\#\text{Time}_{\texttt{Ours}}$ report the average time costs of *Parsynt* and *AutoLifter*, respectively, and Column $\#\text{Time}_{\texttt{Res}}$ 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 $\#\text{Aux}_{\texttt{SP}}$ 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*, $\text{Time}_{\texttt{Res}}$) 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 *Parsyn21*, 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 $ls\,(xs +\!\!+ [v]) = v \geq (max\ xs)$. 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 $ls\,(l_1 +\!\!+ l_2)$ is always equal to $(ls\ l_2) \wedge (max\ l_1 \leq max\ l_2)$. *AutoLifter* can generate this simpler solution, because it finds auxiliary values by enumerating programs in $\mathcal{L}_{aux}$ instead of transforming the source code of the original program.

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 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. Similarly 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 $DPASyn_{=}$) for establishing a fair comparison and an enhanced configuration (denoted as $DPASyn_{+}$) for better revealing the performance of *DPASyn*.

- $DPASyn_{=}$ uses the same language as *AutoLifter* to specify the program space.
- $DPASyn_{+}$ 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 $\mathcal{L}_{comb}$ 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 $DPASyn_{+}$ by (1) adding operators max and min and (2) excluding those operators that are not used.

Note that the comparison between *AutoLifter* and $DPASyn_{+}$ favors the latter, because $DPASyn_{+}$ 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*.

Solver	#Solved	$\text{Time}_{\texttt{Base}}$	$\text{Time}_{\texttt{Ours}}$	$\text{Time}_{\texttt{Res}}$
AutoLifter	**33**/39	8.861		×1.000
$DPASyn_{=}$	15/39	10.32	**4.685**	×1.019
$DPASyn_{+}$	21/39	27.74	**3.951**	×1.044

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.

Table 7. The Results of Comparing *AutoLifter* with $AutoLifter_{\mathsf{OE}}$

Solver	D&C				Single-pass			
	#Solved	$\text{Time}_{\mathtt{Base}}$	$\text{Time}_{\mathtt{Ours}}$	$\text{Time}_{\mathtt{Res}}$	#Solved	$\text{Time}_{\mathtt{Base}}$	$\text{Time}_{\mathtt{Ours}}$	$\text{Time}_{\mathtt{Res}}$
AutoLifter	**29**/36	30.79		×1.000	**33**/39	8.861		×1.000
$AutoLifter_{\mathsf{OE}}$	13/36	3.329	**1.010**	**×0.985**	28/39	5.563	**4.141**	**×0.954**
Solver	Longest Segment				Segment Tree			
	#Solved	$\text{Time}_{\mathtt{Base}}$	$\text{Time}_{\mathtt{Ours}}$	$\text{Time}_{\mathtt{Res}}$	#Solved	$\text{Time}_{\mathtt{Base}}$	$\text{Time}_{\mathtt{Ours}}$	$\text{Time}_{\mathtt{Res}}$
AutoLifter	**7**/8	22.87		×1.000	**13**/13	43.30		×1.000
$AutoLifter_{\mathsf{OE}}$	**7**/8	46.79	**22.87**	**×0.968**	8/13	48.69	**39.66**	×1.011
Solver	Total							
	#Solved	$\text{Time}_{\mathtt{Base}}$	$\text{Time}_{\mathtt{Ours}}$	$\text{Time}_{\mathtt{Res}}$				
AutoLifter	**82**/96	23.27		×1.000				
$AutoLifter_{\mathsf{OE}}$	56/96	16.36	**10.83**	**×0.971**				

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 $AutoLifter_{\mathsf{OE}}$) 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. Similarly 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, $AutoLifter_{\mathsf{OE}}$ 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 *maximum segment product (msp)* [Bird 1989a], which is an advanced version of *mss* (Section 4.1). Given list $xs[1 \ldots n]$, 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 $[-1, -5] +\!\!+ [-3, 0]$ 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 s (where 113.9 s are used by bounded model checking) and finds an auxiliary program as follows:

$$aux\ xs \triangleq \big(max\ (scanl\ (\times)\ xs),\ max\ (scanr\ (\times)\ xs),\ min\ (scanl\ (\times)\ xs),\ min\ (scanr\ (\times)\ xs),\ head\ (scanr\ (\times)\ xs)\big).$$

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 $min\ s + max\ s > length\ s$ 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 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 xs_L, xs_R and integer v satisfying $v < min\ xs_L \wedge v \leq min\ xs_R$,

$$(orig \triangle aux)\ (xs_L +\!\!+ [v] +\!\!+ xs_R) = comb\ \big(v, \big((orig \triangle aux)\ xs_L, (orig \triangle aux)\ xs_R\big)\big),$$

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 $aux\ xs \triangleq (length\ xs, max\ xs)$ and a correct combinator *comb* in 100.0 s. 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 $max\ xs_L \geq max\ xs_R$, as follows:

$$comb\ (v, (res_L, res_R)) \triangleq \begin{cases} \max(lsp_R, \min(len_L + len_R + 1, v + max_L - 1)) & v + max\ xs_L > length\ xs_L + 1, \\ \max(lsp_L, lsp_R) & otherwise \end{cases}$$

where res_L is unfolded to $(lsp_L, (len_L, max_L))$, res_R is unfolded to $(lsp_R, (len_R, max_R))$

assuming $max\ xs_L \geq max\ xs_R$.

Case 1: $(max\ xs_L \geq max\ xs_R) \wedge (v + max\ xs_L > length\ xs_L + 1)$. There are only three possible cases for the longest valid segment: the longest valid segment s_L in xs_L, the longest valid segment s_R in xs_R, or the longest valid segment s_v including element v. In this case, s_L is no longer than s_v, because segment $xs_L +\!\!+ [v]$ is valid under the condition that $v + max\ xs_L > length\ xs_L + 1$. Therefore, the longest valid segment of the whole list must be the longer one between s_R and s_v. Since the length of s_R is known as lsp_R, the remaining task is to get the length of s_v.

An observation is that segment $xs_L +\!\!+ [v]$ already achieves the maximum possible $min\ s + max\ s$ among segments including v, because (1) $min\ s$ 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 $max\ xs_L \geq max\ xs_R$. Therefore, s_v 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, which means the length of s_v must be $\min(len_L + len_R + 1, v + max_L - 1)$.

Case 2: $(max\ xs_L \geq max\ xs_R) \wedge (v + max\ xs_L \leq length\ xs_L + 1)$. In this case, s_v 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 $\max(lsp_L, lsp_R)$. This property can be proved in two steps. First, s_v must be no longer than xs_L, as shown by the following derivation:

$$length\ s_v \leq v + max\ s_v - 1 \leq v + max\ xs_L - 1 \leq length\ xs_L.$$

The first inequality uses the fact that v is the minimum of the whole list, the second inequality uses the fact that $max\ xs_L$ is the maximum of the whole list under the condition that $max\ xs_L \geq max\ xs_R$, and the third inequality uses the condition that $v + max\ xs_L \leq length\ xs_L + 1$.

Second, since s_v 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_v. As shown by the derivation below, s' must be valid as well. Therefore, s' is no longer than s_L, implying that s_v is no longer than s_L,

$$min\ s' + max\ s' \geq v + max\ s_v > length\ s_v = length\ s'.$$

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 then 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 article, 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 vs. 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 as follows:

— 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).

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 article only to explain the effectiveness of *AutoLifter* and clarify the boundary of *AutoLifter*.

Table 8. The Largest Outputs from an Integer List Whose Length $\leq n$ and Element Integers $\in [-m, m]$

Program	*length*	*min*	*sndmin*	*sum*	*mps*	*mss*	*msp* (Section 8.6)
Max Output	n	m		$n \times m$			n^m

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, then 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 $2^a 3^b 5^c$ 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 *sndmin* (Section 3.1), the resulting program generated from the natural implementation of *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 *AutoLifter* and Acar et al. [2005]'s approach have their advantages in automatic incrementalization. *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 *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 *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, *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 *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 *sndmin* (where the first minimum is required as an auxiliary value). Compared with these approaches, *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 *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, *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. *DraydSynth* 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 the following 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 *sndmin* (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 article, we study the problem of applying D&C-like algorithmic paradigms and aim to address the limitation of previous transformation-based approaches that 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 article 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].

APPENDICES

A APPENDIX: PROOFS AND GUARANTEES

This section provides the proofs for the theorems in this article (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 following two facts:

— 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(*orig*, *op*) *of which the mismatch factor is at least* t*, the size-limited unrealizable rate of AutoLifter is bounded as follows:*

$$unreal(orig, op, lim_s) \le 2^w \exp(-t/s_V^{n\cdot w}), \textbf{ where } w \triangleq lim_c \cdot lim_s.$$

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

— Let $\tilde{\varphi}$ be the first *aux* subtask generated from φ, of which the specification is as follows:

$$(orig \triangleleft aux_1)^n\ \overline{a} = (orig \triangleleft aux_1)^n\ \overline{a'} \rightarrow orig\ (op\ (c, \overline{a})) = orig\ (op\ (c, \overline{a'})).$$

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

$$\mathbb{A}(\varphi) \triangleq \{aux \mid \exists comb, \varphi(aux, comb) \wedge \text{size}(aux, comb) \le lim_s\}.$$

Step 1: A sufficient condition. Any program in $\mathbb{A}(\varphi)$ is valid for $\tilde{\varphi}$ by definition. Furthermore, *AutoLifter* will not generate unrealizable subtasks when a program in $\mathbb{A}(\varphi)$ is synthesized from

$\tilde{\varphi}$. Specifically, when such a program is synthesized, *AutoLifter* will only generate three subtasks after $\tilde{\varphi}$ as follows:

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

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 following probability:

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin\mathbb{A}(\varphi)\;\middle|\;|\mathbb{A}(\varphi)|>0\right]. \tag{24}$$

This probability is difficult for direct analysis, because it involves second-order quantification (in the definition of $\mathbb{A}(\varphi)$) 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 $\mathbb{A}(\varphi)$. The conditional probability can be transformed as follows:

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin\mathbb{A}(\varphi)\;\middle|\;|\mathbb{A}(\varphi)|>0\right]\\
=\;&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin\mathbb{A}(\varphi)\wedge|\mathbb{A}(\varphi)|>0\right]\Big/\Pr_{\varphi\sim\mathcal{M}}\left[|\mathbb{A}(\varphi)|>0\right]\\
=\;&\sum_{P\subseteq\mathcal{L}_{aux}}[|P|>0]\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin P\wedge\mathbb{A}(\varphi)=P\right]\Big/\sum_{P\subseteq\mathcal{L}_{aux}}[|P|>0]\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi)=P\right], &(25)\\
\le\;&\frac{\sum_{P\subseteq\mathcal{L}_{aux}}[|P|>0]\left(\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin P\wedge\mathbb{A}(\varphi)=P\right]+(|P|-1)\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi)=P\right]\right)}{\sum_{P\subseteq\mathcal{L}_{aux}}|P|\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi)=P\right]}. &(26)
\end{aligned}$$

The inequality of $(a+c)/(b+c)\ge a/b$ (when $a,b>0,c\ge0,a<b$) is used in the last step.

Let us consider the claim below under the premise that $|P|>0$.

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\notin P\wedge\mathbb{A}(\varphi)=P\right]+(|P|-1)\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi)=P\right]\\
&\qquad=\sum_{aux^*\in P}\Pr_{\varphi\in\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\wedge\mathbb{A}(\varphi)=P\right].
\end{aligned}$$

To prove this claim, let φ be any task satisfying $\mathbb{A}(\varphi)=P$. There are two cases on $\mathcal{S}_{aux}(\tilde{\varphi})$ as follows:

— When $\mathcal{S}_{aux}(\tilde{\varphi})\notin P$, the probability of φ contributes to both sides for $|P|$ times.
— Otherwise, the probability of φ 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 $\mathbb{A}(\varphi)$ is no larger than lim_s by the definition of $\mathbb{A}(\varphi)$. By applying the claim to Formula (26), we further perform the derivation below, where $\mathbb{L}_{\le lim_s}$ denotes

the subspace of $\mathcal{L}_{aux}$ including only those programs of which the size is no larger than lim_s,

$$\begin{aligned}
\text{Formula (26)} &= \sum_{P\subseteq\mathcal{L}_{aux}}\sum_{aux^*\in P}\Pr_{\varphi\in\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\wedge\mathbb{A}(\varphi)=P\right]\Big/\sum_{P\subseteq\mathcal{L}_{aux}}\sum_{aux^*\in P}\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi)=P\right]\\
&= \sum_{aux^*\in\mathbb{L}_{\leq lim_s}}\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\wedge aux^*\in\mathbb{A}(\varphi)\right]\Big/\sum_{aux^*\in\mathbb{L}_{\leq lim_s}}\Pr_{\varphi\sim\mathcal{M}}\left[aux^*\in\mathbb{A}(\varphi)\right]\\
&\leq \max_{aux^*\in\mathbb{L}_{\leq lim_s}}\left(\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\wedge aux^*\in\mathbb{A}(\varphi)\right]\Big/\Pr_{\varphi\sim\mathcal{M}}\left[aux^*\in\mathbb{A}(\varphi)\right]\right), \qquad (27)\\
&= \max_{aux^*\in\mathbb{L}_{\leq lim_s}}\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]. \qquad (28)
\end{aligned}$$

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

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

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

Step 3: Eliminating the output of $\mathcal{S}_{aux}$. A worth noting property of $\mathcal{S}_{aux}$ (Algorithm 4) is that, given a task that has a solution with a size no lager than lim_s, the size of its synthesis result is no larger than $lim_c lim_s$. First, the result of $\mathcal{S}_{aux}$ includes at most 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_s) will be found by OE and be synthesized instead.

Therefore, "any program in $\mathbb{L}_{\leq lim_c lim_s}$ is invalid for $\tilde{\varphi}$ except aux^*" forms a sufficient condition of $\mathcal{S}_{aux}(\tilde{\varphi})=aux^*$. Then, the conditional probability in Formula (28) can be bounded as follows:

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi})\neq aux^*\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]\\
\leq\;&\Pr_{\varphi\sim\mathcal{M}}\left[\exists aux\in\mathbb{L}_{\leq lim_c lim_s}, aux\neq aux^*\wedge\tilde{\varphi}(aux)\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]\\
\leq\;&\sum_{aux\in\mathbb{L}_{\leq lim_c lim_s}}[aux\neq aux^*]\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]\\
\leq\;&2^{lim_s lim_c}\max_{aux\in\mathbb{L}_{\leq lim_c lim_s}}[aux\neq aux^*]\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]. \qquad (29)
\end{aligned}$$

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 $\mathcal{S}_{aux}$ 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 $\mathbb{A}(\varphi)$:

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]=\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\;\middle|\;\exists comb,\varphi(aux^*,comb)\wedge\text{size}(aux^*,comb)\leq lim_s\right].$$

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 $\tilde{\varphi}(aux)$ is determined only by *aux*, while the condition $aux^*\in\mathbb{A}(\varphi)$ 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 follows:

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\;\middle|\;aux^*\in\mathbb{A}(\varphi)\right]=\Pr_{aux}\left[\tilde{\varphi}(aux)\right].$$

Step 5: Bounding the probability of $\tilde{\varphi}(aux)$ using the mismatch factor. Let us first unfold and transform the above probability $\Pr[\tilde{\varphi}(aux)]$ as follows:

$$\begin{aligned}
&\Pr_{aux}\left[\tilde{\varphi}(aux)\right] \\
=&\Pr_{aux}\left[(orig \triangleq aux)^n\ \overline{a} = (orig \triangleq aux)^n\ \overline{a'} \rightarrow orig\,(op\,(c,\overline{a})) = orig\,(op\,(c,\overline{a'}))\right] \\
=&\Pr_{aux}\left[\left(orig^n\ \overline{a} = orig^n\ \overline{a'} \wedge orig\,(op\,(c,\overline{a})) \neq orig\,(op\,(c,\overline{a'}))\right) \rightarrow aux^n\ \overline{a} \neq aux^n\ \overline{a'}\right]. \qquad (30)
\end{aligned}$$

Suppose the mismatch factor of $(orig, op)$ 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 follows:

$$\exists c, \left(orig^n\ \overline{a_i} = orig^n\ \overline{a_i'} \wedge orig\,(op\,(c,\overline{a_i})) \neq orig\,(op\,(c,\overline{a_i'}))\right),$$

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:

$$\begin{aligned}
\Pr_{aux}\left[\tilde{\varphi}(aux)\right] &\leq \Pr_{aux}\left[\bigwedge_{i=1}^{t} aux^n\ \overline{a_i} \neq aux^n\ \overline{a_i'}\right] \leq \prod_{i=1}^{t}\Pr_{aux}\left[aux^n\ \overline{a_i} \neq aux^n\ \overline{a_i'}\right] \\
&\leq \left(1 - \mathrm{pow}(s_V, -lim_c lim_s)^n\right)^t \leq \exp\left(-t/\mathrm{pow}(s_V, n lim_c lim_s)\right).
\end{aligned}$$

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 larger than $lim_s \cdot lim_s$. Under this condition, aux can only output at most $lim_c \cdot lim_s$ auxiliary values, and thus the size of the output domain of aux is no larger than $\mathrm{pow}(s_V, lim_c \cdot lim_s)$.

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

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux)\ \middle|\ aux^* \in \mathbb{A}(\varphi)\right] \leq \exp(-t/s_V^{n\cdot w}), \textbf{where } w \triangleq lim_c \cdot lim_s.$$

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

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{S}_{aux}(\tilde{\varphi}) \neq aux^*\ \middle|\ aux^* \in \mathbb{A}(\varphi)\right] \leq 2^w \exp(-t/s_V^{n\cdot w}).$$

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,

$$2^w \exp(-t/s_V^{n\cdot w}),\ \textbf{where } w \triangleq lim_c \cdot lim_s.$$

□

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., $s_V > 1$), for any constant $\epsilon > 0$, there always exists a constant δ 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 ϵ as follows:*

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

where orig is a random function with type $A \rightarrow V$, op is a random function with type $V \times A^n \rightarrow A$, and 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 ϵ. By taking δ 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 follows:

$$\begin{aligned}
&\Pr_{orig,\, op}\left[\text{the mismatch factor of } \mathsf{LP}(orig, op) < \lfloor \delta \cdot s_A/(n \cdot s_V^{n+1}) \rfloor\right] \\
&= \Pr_{orig,\, op}\left[\text{the mismatch factor of } \mathsf{LP}(orig, op) < 0\right] \\
&= 0 \leq \epsilon.
\end{aligned}$$

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 αs_A and $(1-\alpha)s_A$ values for some constant α, respectively. This construction ensures that the outputs of *orig* on A_1 and A_2 are independent.

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

$$\begin{aligned}
\Pr[\mathcal{E}] &\leq \sum_{v \in V} \Pr\left[num(v) > 2|A_2|/3\right] \\
&\leq \sum_{v \in V} \Pr\left[num(v) > (1 + 1/3)\mathbb{E}\left[num(v)\right]\right] \\
&\leq s_V \cdot \exp\left(-\frac{(1-\alpha)s_A}{21 s_V}\right).
\end{aligned}$$

Here the first step uses the union bound, the second step uses the facts that $\mathbb{E}[num(v)] \leq |A_2|/s_v$ 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 $m = \lfloor |A_1|/(2n) \rfloor$ 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 following conditions for a given value $c \in C$:

$$orig^n\ \overline{a} = orig^n\ \overline{a'} \qquad orig\,(op\,(c, \overline{a})) \neq orig\,(op\,(c, \overline{a'})) \qquad op\,(c, \overline{a}) \in A_2 \qquad op\,(c, \overline{a'}) \in A_2.$$

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 *orig* on A_1, and the outputs of *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 *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 τ is an arbitrary constant in $(0, 1)$,

$$\begin{aligned}
&\Pr\left[|S'| \le (1-\tau)\cdot\gamma\cdot s_A/(n\cdot s_V^n) \;\middle|\; \neg\mathcal{E}\right] \\
\le\ &\Pr\left[|S'| \le (1-\tau)\mathbb{E}[|S'|] \;\middle|\; \neg\mathcal{E}\right] \\
\le\ &\exp\left(-\tau^2/2\cdot\mathbb{E}[|S'|]\right) \\
\le\ &\exp\left(-\frac{\tau^2}{2}\cdot\gamma\cdot\frac{s_A}{n\cdot s_V^n}\right).
\end{aligned}$$

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

$$\begin{aligned}
&\Pr\left[\text{the mismatch factor of } \mathsf{LP}(orig, op) \le (1-\tau)\cdot\gamma\cdot s_A/s_V^n\right] \\
\le\ &\Pr\left[|S'| \le (1-\tau)\cdot\gamma\cdot s_A/(n\cdot s_V^{-n}) \;\middle|\; \neg\mathcal{E}\right] + \Pr[\mathcal{E}] \\
\le\ &\exp\left(-\frac{\tau^2}{2}\cdot\gamma\cdot\frac{s_A}{n\cdot s_V^n}\right) + s_V\cdot\exp\left(-\frac{(1-\alpha)s_A}{21 s_V}\right).
\end{aligned}$$

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 follows, resulting in the target lemma:

$$\begin{aligned}
&\le \exp\left(-\frac{\tau^2}{2}\cdot\gamma\cdot t\right) + s_V\cdot\exp\left(-\frac{(1-\alpha)}{21}\cdot s_V\cdot t\right) \\
&\le \exp\left(-\frac{\tau^2}{2}\cdot\gamma\cdot t\right) + \exp\left(-\frac{(1-\alpha)}{21}\cdot t\right) \textbf{ when } t > \frac{210}{1-\alpha} \\
&\le \epsilon \textbf{ when } t > k\cdot\ln(1/\epsilon) \text{ for a large-enough constant } k.
\end{aligned}$$

□

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 ϵ tends to 0 when $s_A/s_V^{w'}$ tends to ∞, where $w' \triangleq n\cdot lim_c\cdot lim_s + n + 1$.*

Proof. To prove this theorem, we need to prove that for any constants $\epsilon, \epsilon' > 0$, the probability for the unrealizable rate to exceed ϵ is at most ϵ' when $s_A/s_V^{w'}$ is large enough as follows:

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

By Lemma A.3, there exists a constant δ such that with a probability of at least $1-\epsilon'$, the mismatch factor on a random lifting problem will be at least $\lfloor\delta\cdot s_A/(n\cdot s_V^{n+1})\rfloor$ as follows:

$$\Pr_{orig,\,op}\left[\text{the mismatch factor of } \mathsf{LP}(orig, op) \ge \lfloor\delta\cdot s_A/(n\cdot s_V^{n+1})\rfloor\right] \ge 1-\epsilon'.$$

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 follows:

$$\mathcal{E} \rightarrow unreal(orig, op, lim_s) \le 2^w\exp(-\lfloor\delta\cdot s_A/(n\cdot s_V^{n+1})\rfloor/s_V^{n\cdot w}), \textbf{ where } w \triangleq lim_c\cdot lim_s.$$

When $s_A/s_V^{w'}$ is larger than a threshold k, the above formula can be simplified as follows:

$$2^w\exp(-\lfloor\delta\cdot s_A/(n\cdot s_V^{n+1})\rfloor/s_V^{n\cdot w}) \le 2^w\exp(-\delta\cdot k/n) \le \epsilon \textbf{ when } k \text{ is large enough.}$$

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

$$\begin{aligned}&\Pr_{orig,op}\left[unreal(orig, op, lim_s) > \epsilon\right]\\ \le &\Pr_{orig,op}\left[unreal(orig, op, lim_s) > \epsilon \mid \mathcal{E}\right] + \Pr_{orig,op}[\neg\mathcal{E}]\\ < &\epsilon' \textbf{ when } s_A/s_V^{w'} \text{ is large enough.}\end{aligned}$$

□

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

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

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

Case 2: aux is not strictly included in any component of aux^*. Since aux^* is formed as a tuple of components (i.e., in the form of $comp_1 \vartriangle \cdots \vartriangle comp_k$), as a sub-program of aux^*, aux must be a tuple of several components in aux^*. So aux must be considered by $\mathcal{S}_{aux}$ before aux^*, since the top-level combination enumerates the number of components from small to large. Consequently, aux should be the synthesis result of $\mathcal{S}_{aux}$ 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 $[1, s_V]$.

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 $\big((xs_L, xs_R), (xs'_L, xs'_R)\big)$ in this sequence satisfies the formula below and (2) every list is used at most once in this sequence,

$$\begin{aligned}(sndmin\ xs_L, sndmin\ xs_R) = (sndmin\ xs'_L, sndmin\ xs'_R)\\ \wedge\ sndmin\ (xs_L +\!\!+ xs_R) \neq sndmin\ (xs'_L +\!\!+ xs'_R).\end{aligned} \tag{31}$$

We can construct such a sequence from those lists formed by $l-2$ integers in range $[3, s_V]$. Specifically, for every such list ys, we construct a tuple $\big((xs_L, xs_R), (xs'_L, xs'_R)\big)$ as follows[11]:

$$\begin{aligned}xs_L &\triangleq [1,2] +\!\!+ ys \qquad & xs_R &\triangleq [1,3] +\!\!+ ys\\ xs'_L &\triangleq [2,2] +\!\!+ ys \qquad & xs'_R &\triangleq [2,3] +\!\!+ ys.\end{aligned}$$

This construction results in a sequence of length $(s_V-2)^{l-2}$, 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.

[11] In this construction, we assume the value of s_V is at least 3.

— 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 $(s_V-2)^{l-2}$. 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 ∞.

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 ith 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 4.55×10^{-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 in our implementation) and δ be the tolerable error rate in the probabilistic guarantee (10^{-3} in the above claim). Then, let $\mathcal{E}$ be the event that the error rate of the synthesis result is more than δ, and let $\mathcal{E}_i$ be the event that a program with an error rate larger than δ is accepted by the verifier in the ith CEGIS iteration.

To get a probabilistic guarantee, our target is to derive an upper bound for $\Pr[\mathcal{E}]$. By the definition, $\mathcal{E}$ happens only when some event $\mathcal{E}_i$ happens, and $\mathcal{E}_i$ happens only when a program with an error rate larger than δ passes $n \cdot i$ random examples, of which the probability is no larger than $(1-\delta)^{n\cdot i}$. Therefore, the following inequality holds:

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

When (n, δ) 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 4.55×10^{-5}.

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 *AutoLifter*.

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.

Procedure. The first paradigm proposed by Zantema [1992] aims at the cases where predicate p is *predict-closed* and *overlap-losed*, defined as follows:

— *prefix-closed* means that for any list satisfying the predicate, all its prefixes must also satisfy the predicate, i.e., $p\ (xs +\!\!+ ys) \rightarrow p\ xs$.

— *overlap-closed* means that for any two overlapped segments satisfying the predicate, their join must also satisfy the predicate, i.e.,

$$(length\ ys > 0 \wedge p\ (xs +\!\!+ ys) \wedge p\ (ys +\!\!+ zs)) \rightarrow p\ (xs +\!\!+ ys +\!\!+ zs).$$

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

```
1  struct Info {
2      bool is_valid;
3      // Variables representing the output of aux.
4  };
5  int lsp(int* A, int n){
6      int res = 0, len = 0;
7      Info info = {/*p [], aux []*/};
8      for (int i = 0; i < n; ++i) {
9          info = /*comb (A[i], info)*/;
10         if (!info.is_valid) {
11             info = {/*p [A[i]], aux [A[i]]*/};
12             if (info.is_valid) {
13                 len = 1;
14             } else {
15                 len = 0, info = {/*p [], aux []*/};
16         } else {
17             len += 1;
18         }
19         res = max(res, len);
20     }
21     return res;
22 }
```

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

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).

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,

$$(p \vartriangleleft aux)\ (xs +\!\!+ [v]) = comb\ \big(v, (p \vartriangleleft aux)\ xs\big).$$

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

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.

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

```
1  struct Info{
2      int res; // Variable representing the output of orig
3      // Variables representing the output of aux
4  }info[N];
5  int rpos[N];
6  int solve(int *A, int n) {
7      int num = 0;
8      for (int i = 0; i < n; i++) {
9          Info now = {/*orig [], aux []*/};
10         while (num > 0 && A[rpos[num]] > A[i]) {
11             now = /*comb (A[rpos[num]], (info[num], now))*/;
12             --num;
13         }
14         num++; rpos[num]=i; info[num]=now;
15     }
16     Info now = {/*orig [], aux []*/};
17     for (int i = num; i > 0; i--) {
18         now = /*comb (A[rpos[i]], (info[i], now))*/;
19     }
20     return now.res;
21 }
```

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

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 as follows:

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

When a new element is inserted, the template merges the last several segments via combinator *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 *comb* again and thus gets the result of the whole list (Line 21).

Reduction to the lifting problem. In this template, combinator *comb* is invoked on an element v and the cared information on two lists xs_L, xs_R, and its task is to return the cared information on $xs_L +\!\!+ [v] +\!\!+ xs_R$. By the definition of the segment partition, all these invocations ensure that v is the leftmost maximum in $xs_L +\!\!+ [v] +\!\!+ xs_R$, i.e., $min\ xs_L > v$ and $min\ xs_R \geq v$.

Therefore, to apply this paradigm, we need to find a combinator *comb* and an auxiliary program *aux* satisfying the following specification:

$$min\ xs_L > v \wedge min\ xs_R \geq v \rightarrow$$
$$(orig \vartriangle aux)\ (xs_L +\!\!+ [v] +\!\!+ xs_R) = comb\ (v, ((orig \vartriangle aux)\ xs_L, (orig \vartriangle aux)\ xs_R)).$$

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

$$op\ (v, (xs_L, xs_R)) \triangleq \begin{cases} xs_L +\!\!+ [v] +\!\!+ xs_R & min\ xs_L > v \wedge min\ xs_R \geq v \\ [] & otherwise \end{cases}.$$

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 *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 xs, 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 as follows:

— Range update (U, a, l, r): set the value of xs_i to $u\ (a, xs_i)$ for each $i \in [l, r]$.
— Range query (Q, l, r): calculate and output the value of $h\ [xs_l, \ldots, xs_r]$.

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:

$$\forall w, u\ (a_0, w) = w \qquad \forall a_1, \forall a_2, w, u\ (a_1, u\ (a_2, w)) = u\ (a_1 \otimes a_2, w).$$

Here, we assume that a_0 and $\otimes$ are directly given for simplicity. In general, finding a_0 and $\otimes$ for a given update program is an isolated synthesis task and thus can be treated separately.

Procedure. A segment tree is a treelike 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 and 12) and then clears it (Line 13).
— `initialize` initializes the information for node *pos* which corresponds to range $[l, r]$. It recurses into two children (Lines 18 and 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 $[ul, ur]$, then the update will be ignored (Line 23). If $[l, r]$ is contained by $[ul, ur]$, then a lazy tag will be put (Line 24). Otherwise, `update` recurses into two children (Lines 26 and 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 $comb_1$

```
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, l, 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.

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,

$$(h \vartriangle aux)\ (xs_L +\!\!+ xs_R) = comb_1\ ((h \vartriangle aux)\ xs_L, (h \vartriangle aux)\ xs_R)$$
$$(h \vartriangle aux)\ (map\ (\lambda w.u\ (a, w))\ xs) = comb_2\ (a, (h \vartriangle aux)\ xs).$$

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

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

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

ACKNOWLEDGEMENT

We thank all anonymous reviewers of this article for their valuable suggestions on this work. We also thank Sirui Lu and Rastislav Bodik for their insightful feedback on this work and generous help in our evaluation.

REFERENCES

Umut A. Acar et al. 2005. *Self-adjusting Computation.* Ph.D. Dissertation. Carnegie Mellon University.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2009. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.* 32, 1 (2009), 3:1–3:53. https://doi.org/10.1145/1596527.1596530

Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically leveraging MapReduce frameworks for data-intensive applications. In *Proceedings of the International Conference on Management of Data (SIGMOD'18).* Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1205–1220. https://doi.org/10.1145/3183713.3196891

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)* Natasha Sharygina and Helmut Veith (Eds.) Lecture Notes in Computer Science, Vol. 8044. Springer, Berlin, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD'13).* 1–8.

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and analysis. arXiv:1904.07146. Retrieved from http://arxiv.org/abs/1904.07146

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17).*

J. L. Bentley. 1977a. Solution to Klee"s rectangle problem.

Jon Louis Bentley. 1977b. Solutions to Klee's rectangle problems. (unpublished), 282–300.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Adv. Comput.* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

Richard Bird. 1989a. Lecture notes in theory of lists.

Richard S. Bird. 1989b. Algebraic identities for program calculation. *Comput. J.* 32, 2 (1989), 122–126. https://doi.org/10.1093/comjnl/32.2.122

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. 1987. Occam's Razor. *Inf. Process. Lett.* 24, 6 (1987), 377–380. https://doi.org/10.1016/0020-0190(87)90114-1

Murray Cole. 1995. Parallel programming with list homomorphisms. *Parallel Process. Lett.* 5 (1995), 191–203. https://doi.org/10.1142/S0129626495000175

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'22)*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 244–259. https://doi.org/10.1145/3519939.3523726

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17).* 540–555. https://doi.org/10.1145/3062341.3062355

Azadeh Farzan and Victor Nicolet. 2021a. Counterexample-guided partial bounding for recursive function synthesis. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV'21). Alexandra Silva and K. Rustan M.*

Leino (Eds.), Part I, Lecture Notes in Computer Science, Vol. 12759. Springer, 832–855. https://doi.org/10.1007/978-3-030-81685-8_39

Azadeh Farzan and Victor Nicolet. 2021b. Phased synthesis of divide and conquer programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. Stephen N. Freund and Eran Yahav (Eds.). ACM, 974–986. https://doi.org/10.1145/3453483.3454089

Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. Albert Cohen and Martin T. Vechev (Eds.). ACM, 572–585. https://doi.org/10.1145/3062341.3062382

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

Jeremy Gibbons. 1996. The third homomorphism theorem. *J. Funct. Program.* 6, 4 (1996), 657–665. https://doi.org/10.1017/S0956796800001908

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'20)*. Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1128–1142. https://doi.org/10.1145/3385412.3385979

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2021. Synthesis with asymptotic resource bounds. arXiv:2103.04188. Retrieved from https://arxiv.org/abs/2103.04188

Kangjing Huang and Xiaokang Qiu. 2022. Bootstrapping library-based synthesis. In *Proceedings of the 29th International Symposium on Static Analysis (SAS'22)* Gagandeep Singh and Caterina Urban (Eds.) Lecture Notes in Computer Science, Vol. 13790. Springer, 272–298. https://doi.org/10.1007/978-3-031-22308-2_13

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'20)*. Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1159–1174. https://doi.org/10.1145/3385412.3386027

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

Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024. Artifact for TOPLAS'24: Decomposition-based synthesis for applying divide-and-conquer-like algorithmic paradigms. (Jan. 2024). https://doi.org/10.5281/zenodo.10472709

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. 253–268. https://doi.org/10.1145/3314221.3314602

Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2023. CBMC: The C bounded model checker. arXiv.2302.02384. Retrieved from https://arxiv.org/abs/2302.02384

Joshua Lau and Angus Ritossa. 2021. Algorithms and hardness for multidimensional range updates and queries. In *Proceedings of the 12th Innovations in Theoretical Computer Science Conference (ITCS'21) (LIPIcs)*, James R. Lee (Ed.), Vol. 185. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 35:1–35:20. https://doi.org/10.4230/LIPIcs.ITCS.2021.35

Woosuk Lee and Hangyeol Cho. 2023. Inductive synthesis of structurally recursive functional programs from non-recursive expressions. *Proc. ACM Program. Lang.* 7, POPL (2023), 2048–2078. https://doi.org/10.1145/3571263

Shu Lin, Na Meng, and Wenxin Li. 2019. Optimizing constraint solving via dynamic programming. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, Sarit Kraus (Ed.). 1146–1154. https://doi.org/10.24963/ijcai.2019/160

Yanhong A. Liu and Scott D. Stoller. 2003. Dynamic programming via static incrementalization. *High. Order Symb. Comput.* 16, 1-2 (2003), 37–62. https://doi.org/10.1023/A:1023068020483

Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic compilation as a functional programming library. *Proc. ACM Program. Lang.* 7, POPL (2023), 455–487. https://doi.org/10.1145/3571209

Kurt Mehlhorn. 1984. *Algorithmic Paradigms*. Springer, Berlin. https://doi.org/10.1007/978-3-642-69900-9_3

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498682

Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the ACM SIGPLAN Conference on*

Programming Language Design and Implementation, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 146–155. https://doi.org/10.1145/1250734.1250752

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. David Grove and Stephen M. Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. 2021. The SyGuS Language Standard Version 2.1. Retrieved from https://sygus.org/assets/pdf/SyGuS-IF_2.1.pdf

Alberto Pettorossi and Maurizio Proietti. 1996. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.* 28, 2 (1996), 360–414. https://doi.org/10.1145/234528.234529

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*. Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 83–98. https://doi.org/10.1145/2048066.2048076

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 65:1–65:28. https://doi.org/10.1145/3133889

Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *Proceedings of theACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. Andrew P. Black and Todd D. Millstein (Eds.). ACM, 909–927. https://doi.org/10.1145/2660193.2660228

Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. Ethan L. Miller and Steven Hand (Eds.). ACM, 153–167. https://doi.org/10.1145/2815400.2815418

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 404–415. https://doi.org/10.1109/ICSE.2017.44

Nicole Schweikardt. 2009. One-pass algorithm. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer US, 1948–1949. https://doi.org/10.1007/978-0-387-39940-9_253

Nicole Schweikardt. 2018. One-pass algorithm. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_253

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. Chandra Krintz and Emery Berger (Eds.). ACM, 326–340. https://doi.org/10.1145/2908080.2908102

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)5, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 287–296. https://doi.org/10.1145/2491956.2462174

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational program synthesis. *Proc. ACM Process. Lang.* 2, OOPSLA (2018), 155:1–155:27.

Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-guided inductive synthesis of recursive functional programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 860–883. https://doi.org/10.1145/3591255

Hans Zantema. 1992. Longest segment problems. *Sci. Comput. Program.* 18, 1 (1992), 39–66. https://doi.org/10.1016/0167-6423(92)90033-8

Received 8 April 2023; revised 8 November 2023; accepted 1 January 2024