Divide and Conquer Divide-and-Conquer

Inductive Synthesis for D&C-Like Algorithmic Paradigms

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

Algorithmic paradigms such as divide-and-conquer (D&C) are proposed to guide the design of efficient algorithms, but applying them to optimize existing programs is difficult. Therefore, many research efforts have been devoted to the automatic application of algorithmic paradigms. However, most existing approaches to this problem are based on deductive methods and thus put significant restrictions on how the original program is implemented. To overcome this limitation, we study the automatic application of paradigms as an *oracle-guided inductive synthesis* problem, where the synthesizer only invokes the original program as a black-box oracle or uses a given verifier to verify the correctness of candidate programs. Such a synthesizer puts no restriction on the original program and thus overcomes the limitation of deductive approaches.

We notice that the application tasks of various paradigms have a similar form as that of D&C. We denote these paradigms as D&C-like paradigms, unify their application tasks into a novel type of synthesis problems, named *lifting problems*, and propose an efficient inductive synthesizer *AutoLifter* for lifting problems. The main challenge of solving lifting problems is from the usually large scale of efficient algorithms. To overcome this challenge, we *divide and conquer* lifting problems. We devise two novel decomposition methods, *component elimination* and *variable elimination*, to soundly divide a lifting problem into simpler subtasks, each tractable with existing inductive synthesizers.

We evaluate *AutoLifter* on 96 programming tasks related to 6 different algorithmic paradigms. *AutoLifter* solves 82/96 tasks with an average time cost of 6.53 seconds, significantly outperforming existing approaches.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Theory of computation** → *Design and analysis of algorithms.*

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], such as divide-and-conquer (D&C),

*Corresponding author

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

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

dynamic programming, greedy, and incrementalization. However, an algorithmic paradigm only tells the general principles of an algorithm class, and implementing these principles for a specific problem is still difficult. For example, D&C only suggests recursively dividing the problem into sub-problems and combining the solutions to the sub-problems into the solution to the original problem. However, how to combine the solutions for a concrete task is totally unknown and up to the developer to discover.

To reduce the burden on the user, many research efforts have been devoted to the automatic application of algorithmic paradigms. These approaches take a possibly inefficient original program as input, apply a specific algorithmic paradigm, and generate a semantically equivalent program with guaranteed efficiency. Typical such approaches include the automatic application of D&C [Farzan and Nicolet 2021b; Morita et al. 2007; Raychev et al. 2015], dynamic programming [Lin et al. 2019], single-pass [Pu et al. 2011], and incrementalization [Acar et al. 2005].

The usually large scale of optimized programs is the major challenge of applying algorithmic paradigms. To cope with this challenge, most existing approaches are deductive in the sense that the source code of the original program is available to the system, and deductive program transformations are used to solve or simplify the task. However, to ensure that these deductive program transformations can be applied successfully, these approaches have strict restrictions on the original program, leading to a significant limitation on their 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 synthesized 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 39.29%-56.90% of the code needed for applying D&C (Section 7.3).

In this paper, we aim to remove the restriction on the original program when automatically applying algorithmic paradigms and thus overcome the limitation of existing approaches. To achieve this, we use an inductive approach and view the problem of applying a paradigm as an *oracle-guided inductive synthesis (OGIS)* problem [Jha and Seshia 2017]. In our work, the goal is to synthesize, in the target algorithmic paradigm, a program that is semantically equivalent to the original one. The synthesizer does not need to access the source code of the original program. Instead, it either (1) invokes the original program as a black-box oracle with some input to obtain the corresponding output, or (2) uses a given verifier to verify the correctness of the synthesized program and obtain counter-examples when the program is incorrect. Such a synthesizer puts no restrictions on the original program, and the user is free to choose any implementation as long as the verifier allows.

The first contribution of this paper is a novel class of synthesis problems, named lifting problems, which unify the application of various paradigms. In contrast to previous approaches that aim at a specific paradigm, we consider a general class of problems that covers at least 7 different paradigms in this paper. We observe that the application task of various paradigms can be viewed as synthesizing (1) a combinator program for combining the solutions of several sub-problems generated by the paradigm and (2) a program specifying necessary auxiliary values for the combination. These paradigms include but not limited to D&C [Cole 1995], 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]. We call such paradigms as *D&C-like* paradigms, unify their synthesis tasks as *lifting problems*, and provide reductions from their application tasks to lifting problems. Through these reductions, every inductive synthesizer for lifting problems can be instantiated as an inductive synthesizer for applying any of the above paradigms.

The second and the most important contribution of this paper is an efficient inductive synthesizer AutoLifter for lifting problems. To address the scalability challenge of synthesizing efficient programs, *AutoLifter* decomposes a lifting problem into a sequence of subtasks, each corresponding to a sub-program of the synthesis target and tractable with existing synthesizers, and solves these subtasks one by one. In other words, we *divide and conquer* the problem of applying D&C-like paradigms.

Decomposing a lifting problem is not straightforward, because sub-programs of the synthesis tasks closely depend on each other, making it difficult to derive precise specifications for individual sub-programs for independent synthesis. To achieve an effective decomposition, *AutoLifter* generates approximate instead of precise specifications in some subtasks. Consequently, it ensures only the soundness of the synthesis result but sacrifices the completeness. *AutoLifter* may generate an unrealizable subtask (i.e., a subtask without any valid solution) from a realizable lifting problem, leading to a failed synthesis. However, despite the theoretical incompleteness, *AutoLifter* performs well in our evaluation: it never generates unrealizable subtasks from a realizable lifting problem in our dataset. We analyze this phenomenon and ascribe it to two factors.

- First, a domain property of practical lifting problems, named the *compressing property*, makes the approximate specification close enough to the precise one.
- Second, the preference of *AutoLifter* on smaller solutions, which matches the principle of *Occam's Razor*, helps avoid incorrect sub-programs that may lead to unrealizable subtasks.

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

- *AutoLifter* solves 82 out of 96 tasks with an average time cost of 6.53 seconds, significantly outperforming existing synthesizers that can be applied to lifting problems. Among solved tasks, the largest result includes 157 AST nodes and is found by *AutoLifter* in 10.2 seconds.
- *AutoLifter* significantly outperforms a specialized synthesizer for single-pass programs and offers competitive, or even better, performance to an existing deductive synthesizer for D&C programs that requires more input from users.

Besides, we conduct a case study using two tasks in our dataset, which shows the advantage of inductive synthesis and the ability of *AutoLifter* on solving tasks difficult for human programmers.

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

- We introduce a novel class of synthesis problems named lifting problems (Section 3) and reduce the application of various algorithmic paradigms to lifting problems (Section 5).
- We propose an efficient inductive solver named *AutoLifter* for lifting problems (Section 4), which decomposes lifting problems into subtasks tractable by existing synthesizers.
- We implement *AutoLifter* (Section 6) and evaluate it on a dataset of 96 related tasks (Section 7). The results demonstrate the advantage of *AutoLifter* compared to existing approaches.

2 OVERVIEW

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

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

Fig. 1. Second minimum.

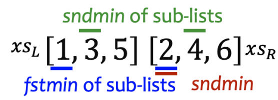

Fig. 2. An example of calculating *sndmin*.

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

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.

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, quite inefficient.

To optimize the natural implementation, let us consider the application of D&C, a paradigm widely used for optimization. In general, D&C decomposes a task into subtasks of the same type but with smaller scales. For tasks on lists, a standard way 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 shown below.

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

In this example, the first minimum of each sub-list is a valid auxiliary value, and the corresponding (*aux*, *comb*) are 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* 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 the sub-list. When the sub-list contains only one element, the original program and the *aux* are directly applied. Otherwise, *dac* is recursively called on the two

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	$N_{\mathbb{Z}} + N_{\mathbb{Z}} \mid \mathit{min}\ N_{\mathbb{L}}$
		$\mid$	$\mathit{max}\ N_{\mathbb{L}} \mid \mathit{sum}\ N_{\mathbb{L}}$
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 5a), and the output of *comb* can also be a tuple since *comb* usually needs to calculate multiple values (Figure 5b).

halves of the sub-list, and the results are combined through *comb*. Note that although *aux* is only applied 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 shown by the procedure above, 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

To address the difficulty in manual optimization, our paper aims to automate the application of D&C. Concretely, our approach takes the original program *sndmin* (Figure 1) as the input, produces proper programs *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)** To ensure the resulting program correctly calculates the second minimum, we use Formula 1 as the specification for synthesizing *aux* and *comb*.
- **(Efficiency)** To ensure an efficient D&C program ($O(n/p)$-time in parallel), we apply the SyGuS framework [Alur et al. 2013] and constrain the space of candidate solutions to include only *comb* that runs in $O(1)$ time and *aux* that runs in $O(1)$ time on singleton lists.
  In this example, we consider a toy solution space (Figure 5) simplified from the one in our implementation (Section 6). 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 the total size of these two programs can be large in real-world algorithmic problems (up to 157 AST nodes in our dataset). General program synthesis approaches that handle relational specifications such as enumerative synthesis [Alur et al. 2013] and relational synthesis [Wang et al. 2018] do not scale up to solve most of the problems in our dataset. On the other hand, most other scalable synthesis algorithms [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 a single program and require obtaining input-output examples, and thus cannot work for two programs in a relational specification.

2.3 An Existing Approach and Its Limitation

Parsynt [Farzan and Nicolet 2017, 2021b] is a state-of-the-art synthesizer specifically designed for D&C. To solve the scalability challenge, *Parsynt* treats the original program as a white box and uses syntax-based program transformations to derive *aux* from the original program. After *aux* is derived, only *comb* is unknown and can be synthesized using existing synthesizers. In this procedure, the full definition of *aux* is derived to help synthesize *comb* though *aux* is invoked only on singleton lists in the D&C template (Figure 4).

To apply syntax-based transformations, *Parsynt* requires the original program to be a single-pass program that enumerates each element in the input list only once. Figure 6 shows a single-pass implementation of *sndmin*, where the program is formed by a loop visiting each element in the input list *xs* once. This program takes the first minimum as an auxiliary value and updates the second minimum using the property that, each time when 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*.

As we can see, to correctly implement *sndmin* as single-pass, one already has to include the first minimum as an auxiliary value and cope with the update of the first and second minimums. In other words, most of the difficult work in applying D&C has been finished while implementing this single-pass program, making the help provided by *Parsynt* limited. In the dataset we used for evaluation, the auxiliary values required for a single-pass implementation account for 39.29%-56.90% of those required by D&C (Section 7.3). Moreover, implementing single-pass programs is also 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, our approach *AutoLifter* views this problem as an OGIS problem and treats the original program as a black-box oracle. *AutoLifter* only invokes the program with some input to obtain the corresponding output or invokes a given verifier to verify the correctness of a candidate program. In this way, *AutoLifter* puts no restriction on the original program and can accept any implementation as the input, such as the one in Figure 1.

However, the scalability challenge resurfaces under the inductive setting. We can no longer directly extract *aux* using syntax-based transformations (as *Parsynt* does) because such transformations for general programs (or black-box oracles) do not exist. In this paper, we explore another direction to cope with the scalability challenge by answering the following question.

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

Our answer is positive. We propose two decomposition methods, named *component elimination* and *variable elimination*, to derive specifications for sub-programs of the synthesis target. By applying these methods, we can first synthesize a part of the synthesis target using the derived specification and then substitute the obtained part into the original specification to synthesize the remainder. In this way, we *divide and conquer* the synthesis of D&C programs, greatly reducing the scale of the program to be synthesized in each step.

Figure 7 demonstrates the workflow of *AutoLifter*, which synthesizes through an interaction between a decomposition system and two inductive synthesizers. Given a synthesis task, the

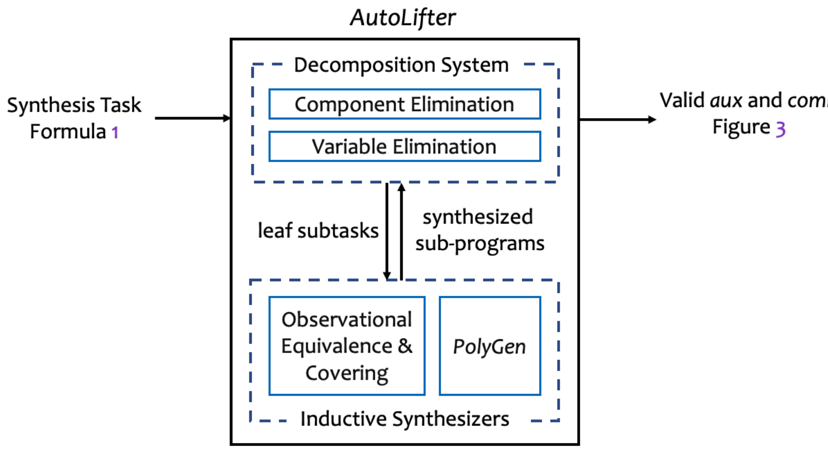

Fig. 7. The workflow of *AutoLifter*.

decomposition methods are applied to decompose the task into a series of *leaf* subtasks (i.e., subtasks that cannot be further decomposed), each with a smaller scale and a simpler form. These subtasks are solved by existing inductive synthesizers, and the results are collected for generating subsequent subtasks and constructing the final result.

For ease of understanding, we introduce only a part of component elimination that is enough for the *sndmin* example. The omitted details will be supplied in Section 4.2.

Component elimination. The original specification (Formula 1) raises two requirements, calculating the expected output (defined by *sndmin*) and calculating the auxiliary values (defined by *aux*). Component elimination aims at separating these two requirements into two subtasks and synthesizes a part of the synthesis target $(aux, comb)$ from each of them.

Concretely, according to the specification, the output of *comb* is a pair of two values, corresponding to the outputs of *sndmin* and *aux*, respectively. Therefore, the form of *comb* can be assumed as $comb\ (res_L, res_R) \triangleq (comb_1\ (res_L, res_R), comb_2\ (res_L, res_R))$ without loss of generality. In the solution shown in Figure 3, $comb_1$ and $comb_2$ correspond to the expressions bound to *csmin* and *caux*, respectively. Then, we can derive a specification involving only *aux* and $comb_1$ and thus synthesizes $(aux, comb_1)$ and $comb_2$ in two sequential subtasks. In this way, a component of *comb* (i.e., $comb_2$) is eliminated from the original specification in the first subtask.

In the first subtask, the specification derived for $(aux, comb_1)$ is as follows. 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.

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

This specification has the same form as the original one (Formula 1) except that only the first component (*sndmin*) of the pair (*sndmin'*) is considered on the left-hand side. In comparison, the scale of this subtask is smaller since only $comb_1$ but not the whole *comb* is considered. One can verify that this specification (Formula 2) is satisfied by taking *aux* as $aux\ xs \triangleq min\ xs$ and taking $comb_1$ as the expression bound to *cmin* in Figure 3, i.e., $comb_1\ ((smin_L, min_L), (smin_R, min_R)) \triangleq \min(smin_L, smin_R, \max(min_L, min_R))$.

After the solution above is synthesized, the second subtask is derived by putting this solution into the original specification, as shown below.

- Let us start with the original specification (Formula 1).

$$sndmin'\ (xs_L \mathbin{+\!\!+} xs_R) = comb\ (sndmin'\ xs_L, sndmin'\ xs_R)$$
$$\textbf{where } sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs)$$

- The intermediate formula below can be obtained by (1) unfolding $sndmin'$ and $comb$ and (2) substituting aux and $comb_1$ with their synthesis results, respectively.

$$\begin{aligned}\Big(sndmin\ (xs_L \mathbin{+\!\!+} xs_R), min\ (xs_L \mathbin{+\!\!+} xs_R)\Big)&\\ = \Big(\min\Big(&sndmin\ xs_L, sndmin\ xs_L, \max(min\ xs_L, min\ xs_L)\Big),\\ &comb_2\ \Big((sndmin\ xs_L, min\ xs_L), (sndmin\ xs_R, min\ xs_R)\Big)\Big)\end{aligned}$$

- Both sides of this equation are a pair of values, and the equality between the first components has already been established by the first subtask. Therefore, $comb_2$ can be synthesized from only the equality between the second components, resulting in the specification below.

$$\boxed{min\ (xs_L \mathbin{+\!\!+} xs_R) = comb_2\ \Big((sndmin\ xs_L, min\ xs_L), (sndmin\ xs_R, min\ xs_R)\Big) \qquad (3)}$$

The two subtasks generated here will be dealt with differently, as shall be introduced later. The first one (Formula 2) will be further decomposed by the other method, *variable elimination*. The second one (Formula 3) will be directly used for synthesis without further decomposition; in other words, it is a leaf subtask of decomposing the original specification (Formula 1). In this section, we include all generated leaf subtasks in framed boxes to distinguish them from the other subtasks.

We have seen the core idea of component elimination, which aims at separating the two requirements raised by the original specification (Formula 1). However, the above decomposition procedure does not consider the possibility that auxiliary values themselves may require more auxiliary values. Imagine that there is another original program that requires the second minimum as an auxiliary value, and *sndmin* is successfully synthesized from the first subtask. At this time, no valid combinator exists for the corresponding second subtask because the synthesized auxiliary value (i.e., the second minimum) still cannot be calculated in D&C unless some extra auxiliary value (i.e., the minimum) is introduced.

This issue can be solved by allowing new auxiliary values to be introduced in the second subtask of component elimination. Then, the second subtask takes the same form as a lifting problem and can be recursively decomposed by component elimination for finding extra auxiliary values. This procedure stops when no new auxiliary value is needed (i.e., an empty auxiliary program is synthesized from the first subtask). The details on this point can be found in Section 4.2.

Variable elimination. Two simpler tasks are obtained after component elimination, but the first one (Formula 2) still involves two variables, $comb_1$ and aux. Variable elimination derives a task involving only aux and thus separates the synthesis of aux and $comb_1$; 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).

$$xs_L\ [1,3,5], [2,4,6]\ xs_R \qquad xs'_L\ [1,3,5], [1,4,6]\ xs'_R \qquad (4)$$

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 $xs_L \mathbin{+\!\!+} xs_R$ (which is 2) differs from that of $xs'_L \mathbin{+\!\!+} xs'_R$ (which is 1). Consequently, a conflict can be derived after substituting these two pairs into the specification of $comb_1$ (Formula 2) when aux is not involved: (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.

Therefore, a necessary condition for a valid aux is to ensure the existence of a function for $comb_1$ to implement. In other words, when the expected outputs (i.e., the second minimums of the combined lists) are different, some parts of the inputs (the second minimums or the auxiliary values on the two halves) must also be different. Formally, given two arbitrary pairs of lists, (xs_L, xs_R) and (xs'_L, xs'_R), the following specification needs to be satisfied.

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

To make the constraint on aux clear, we transform this specification to an equivalent form (shown below) by unfolding $sndmin'$ and performing equivalence transformations. This specification does not involve $comb_1$ and is used as a subtask to synthesize aux.

$$\begin{aligned}
&(\mathit{sndmin}\ xs_L, \mathit{sndmin}\ xs_R) = (\mathit{sndmin}\ xs'_L, \mathit{sndmin}\ xs'_R) \\
&\qquad \wedge\ \mathit{sndmin}\ (xs_L \mathbin{+\!\!+} xs_R) \neq \mathit{sndmin}\ (xs'_L \mathbin{+\!\!+} xs'_R) \\
&\qquad\qquad \rightarrow (\mathit{aux}\ xs_L, \mathit{aux}\ xs_R) \neq (\mathit{aux}\ xs'_L, \mathit{aux}\ xs'_R)
\end{aligned} \tag{5}$$

One can verify that the above specification is satisfied by taking aux as $aux\ xs \triangleq min\ xs$. After this program is synthesized, we can put it into the specification of $(aux, comb_1)$ (Formula 2) and obtain a subtask for a corresponding $comb_1$, as shown below.

$$\mathit{sndmin}\ (xs_L \mathbin{+\!\!+} xs_R) = \mathit{comb}_1\ \big((\mathit{sndmin}\ xs_L, \mathit{min}\ xs_L), (\mathit{sndmin}\ xs_R, \mathit{min}\ xs_R)\big) \tag{6}$$

Synthesis from leaf tasks. After applying the above two decomposition methods, the original synthesis task (Formula 1) is decomposed into three consecutive leaf tasks (Formulas 5, 6, and 3). This decomposition not only reduces the synthesis scale but also greatly simplified the form of specifications, making leaf subtasks tractable with existing inductive synthesis techniques.

We apply the framework of counter-example guided inductive synthesis (CEGIS) [Solar-Lezama 2013] to solve these leaf tasks. 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 three leaf tasks, the tasks for $comb_1$ and $comb_2$ (Formulas 6 and 3) are in the same form and no longer relational. Input-output examples can be easily extracted from examples of these tasks, for example, $comb_1$ is required to output 2 from input $((3, 1), (4, 2))$ under example $(xs_L \triangleq [1, 3, 5], xs_R \triangleq [2, 4, 6])$ of Formula 6. As a result, synthesis algorithms relying on input-output examples are available for these two tasks, and we use a state-of-the-art synthesizer *PolyGen* [Ji et al. 2021] in our implementation.

In contrast, the task for aux (Formula 5) is still relational, where the outputs of aux on different inputs are involved, making input-output examples unavailable. Even so, this task fits into the scope of another efficient synthesis algorithm named *observational equivalence (OE)* [Alur et al. 2013]. OE is configured by an input set. It enumerates programs from small to large by combining existing programs with language constructs and prunes off duplicated programs that output the same on the given input set. Here, whether *aux* satisfies an example depends only on its outputs on xs_L, xs_R, xs'_L, xs'_R so that OE can be applied by including all these inputs into the input set.

Besides, we also integrate a specialized pruning method, named *observational covering*, into OE to further speed up the synthesis. This method focuses on the cases requiring multiple auxiliary values and utilizes the relation between *aux* and each auxiliary value it calculates. The details of this method 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 greatly reduces the difficulty of synthesizing *comb*. As we can see, after the full definition is synthesized, the subsequent subtasks for *comb* are no longer relational and can be solved easily.
- Neither *PolyGen* nor OE can be directly applied to the original problem (Formula 1) without the decomposition. For *PolyGen*, neither input-output examples of *aux* nor those of *comb* can be extracted from Formula 1; and for OE, the validness of *comb* in the original problem relies on its output on an unknown input produced by *aux*.

Properties of *AutoLifter*. The decomposition methods of *AutoLifter* are *sound*, i.e., any solution constructed from valid sub-programs for the decomposed leaf subtasks are sound. In each decomposition, the second subtask is always obtained by putting the result of the first subtask into the original specification. Therefore, the original specification must be satisfied when the second subtask is solved successfully.

However, the decomposition methods used by *AutoLifter* is *not complete*, possibly decomposing a realizable task (i.e., a task whose valid solution exists) into unrealizable subtasks. This is because, in both decomposition methods, the specification derived for the first subtask is weaker than the original one, and thus it is possible to synthesize a program in the first subtask that makes the second subtask unrealizable. For example, the subtask for *aux* (Formula 5) only ensures that a function exists for $comb_1$ to implement but does not ensure that such an implementation exists in the program space ($\mathcal{L}^{ex}_{comb}$, Figure 5b). One can verify that $(min\ xs) + (min\ xs)$ is also valid for this subtask, but a corresponding combinator does not exist in $\mathcal{L}^{ex}_{comb}$.

To deal with incompleteness, one possible way is to combine the decomposition system with a backtracking mechanism, which synthesizes alternative *aux* if previous ones lead to unrealizable subtasks. However, we found such backtracking is not needed because this theoretical incompleteness of *AutoLifter* is never encountered in our evaluation; in other words, *AutoLifter* never generates unrealizable subtasks from realizable tasks in our dataset. In the remainder of this subsection, we shall intuitively discuss why this happens on this *sndmin* example.

To solve the example task, the key is to ensure that exactly *min xs* is synthesized from the subtask for *aux* (Formula 5) because the subsequent two subtasks are both determined by this result. *AutoLifter* achieves this through a combined effect between the enumeration-based synthesizer (mainly OE) and the program space ($\mathcal{L}^{ex}_{aux}$, Figure 5a).

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 5) as well, the property of *Occam's razor* [Blumer et al. 1987; Ji et al. 2021] exists here: the intended solution *min xs* is the minimal program in this category. Because *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*, and the *compressing* property of these programs makes the specification for *aux* (Formula 5) strong enough to exclude them all. As a side effect of ensuring an efficient D&C program, program space $\mathcal{L}^{ex}_{aux}$ includes only

Table 1. Counter-examples of *sum xs* and *max xs* for Formula 5.

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	$([0,2],[0,2])$	$([0,2],[1,2])$	$(2,2) = (2,2) \wedge 0 \neq 1 \rightarrow (2,2) \neq (2,2)$

programs mapping a list (whose size is unbounded) to a constant-sized tuple of integers. These programs *compress* a large input space to a much smaller output space[1] and thus frequently output the same on different inputs. Consequently, an incorrect program in $\mathcal{L}^{ex}_{aux}$ can hardly satisfy the specification (Formula 5), which requires *aux* to output differently 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 5, and the corresponding counter-examples are listed in Table 1.

Note that the specification for *aux* (Formula 5) can 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 4.4.

- First, we prove that the probability for *AutoLifter* to be incomplete converges to 0 under a probabilistic model where programs are modeled as independent random functions with the compressing property (Theorem 4.13, Corollary 4.17).
- 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 4.19).

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 similar algorithmic 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 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 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 shown below.

$$sndmin'\ (append\ xs\ v) = comb\ (sndmin'\ xs, v), \textbf{ where } sndmin'\ xs \triangleq (sndmin\ xs, aux\ xs) \quad (7)$$

A valid solution to this specification is shown in Figure 8. Similar to the D&C case, the auxiliary value is still the minimum element of the list, and the combinator updates both the second minimum and the first minimum based on 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.

$aux\ xs = min\ xs$
$comb\ ((smin_{pre}, aux_{pre}), v) =$
$\quad \textbf{let}\ csmin = \min(smin_{pre}, \max(aux_{pre}, v))\ \textbf{in}$
$\quad\quad \textbf{let}\ caux = \min(aux_{pre}, v)\ \textbf{in}$
$\quad\quad\quad (csmin, caux)$

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

[1]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 8.

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 paper, we regard a type as a set of values of the type and use the two terms interchangeably. To distinguish between types and values, we use uppercase letters such as A, B to denote types, and lowercase letters (or words) such as $a, xs, func$ 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, $func_1 \vartriangle func_2$ 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.

$$(func_1 \vartriangle func_2)\ x \triangleq (func_1\ x, func_2\ x) \qquad (func_1 \times func_2)\ (x_1, x_2) \triangleq (func_1\ x_1, func_2\ x_2)$$
$$func^n\ (x_1, \dots, x_n) \triangleq (func\, x_1, \dots, func\, x_n)$$

Problem definition. Given an original program *orig* over some data-structure type and an operator that creates an instance a from some other instances $a_1, \dots, a_n$ 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* $a_1, \dots,$ *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.

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 $\overline{a} \in A^n$:

$$(orig \vartriangle aux)\ (op\ (c, \overline{a})) = comb\ (c, (orig \vartriangle aux)^n\ \overline{a}) \tag{8}$$

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 in 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 7).

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

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.

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	Formula 7			$op\ (c, (xs)) \triangleq append\ xs\ c$	1	`Int`

A lifting problem is defined as a syntax-guided synthesis (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 7) must run in $O(1)$ time per change if $\mathcal{L}_{comb}$ includes only programs running in constant time.

In this paper, we assume that $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ are implicitly given and denote a lifting problem as $\mathsf{LP}(orig, op)$. 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.

$$\begin{tikzcd} A \arrow[r, "orig \triangle aux"] & B \\ C \times A^n \arrow[u, "op"] \arrow[r, "id \times (orig \triangle aux)^n"] & C \times B^n \arrow[u, "comb"'] \end{tikzcd}$$

$$\boxed{B \triangleq B_{orig} \times B_{aux}}$$

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. In detail, *id* is the identity function, and B denotes the output type of $orig \triangle aux$. This diagram shows that $orig \triangle aux$ 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 8), the left-hand side explicitly constructs an intermediate A-element via *op* and then 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 Algorithmic Paradigms. As mentioned before, the application of many algorithmic paradigms can be reduced to lifting problems. The reductions for paradigms other than D&C and incrementalization can be found in Section 5. Given a reduction, every synthesizer for lifting problems can be instantiated as a synthesizer for applying the corresponding paradigm. In practice, to apply a certain algorithmic paradigm, the end user only needs to pick up the corresponding instantiated synthesizer and provide the original program, and then the instantiated synthesizer will automatically generate a semantically equivalent program in the target paradigm.

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 4.4 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 all the integers within the segment to get the segment sum. The *mss*

```
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

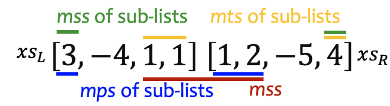

Fig. 10. An example of calculating *mss*.

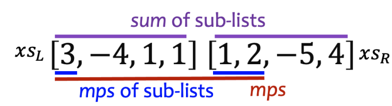

Fig. 11. An example of calculating *mps*.

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

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

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

problem is to find, for a given list, the greatest sum we can get among all possible segments. A natural implementation of *mss* (Figure 9) enumerates all possible segments, calculates the sums of their elements, and returns the maximum. This program runs in $O(n^3)$ time on a lift of length n and thus is quite inefficient.

D&C can be applied to optimize this natural implementation. However, similar to the second minimum example, if we divide the input list into two halves, the *mss* of the whole list cannot be calculated from those of the two halves. In the case shown in Figure 10, the segment with the maximum sum of the left half is the prefix list [3], 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 are also calculated during divide-and-conquer, and the same issue shall occur again: no corresponding combinator exists unless new auxiliary values are introduced. Figure 11 demonstrates such a case, where the *mps* of the whole list covers the full left half, and its sum cannot be produced using only *mps*, *mts*, and *mss* of the two halves. Here, we can introduce the sum of integers in the list as a supplementary auxiliary value to enable the calculation of *mps* and *mts*. In this way, the *mps* of the whole list in Figure 11 can be produced by adding up the sum of the left half and the *mps* of the right half.

The task of applying D&C to *mss* can be regarded as a lifting problem $\mathsf{LP}(mss, op)$ for operator $op\ ((), (xs_L, xs_R)) \triangleq xs_L +\!\!+\ xs_R$. Its specification is shown below.

$$(mss \vartriangle aux)\ (xs_L +\!\!+\ xs_R) = comb\ ((), (mss \vartriangle aux)^2\ (xs_L, xs_R)) \tag{9}$$

For simplicity, in this example, we continue using $\mathcal{L}^{ex}_{comb}$ (Figure 5b) and extend $\mathcal{L}^{ex}_{aux}$ (Figure 5a) 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 6.1, Section 6).

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. The main part of the solution is expressions to produce the components in the tuples. These expressions are labeled for later reference.

4.2 Decomposition System

AutoLifter includes two decomposition methods, *component elimination* and *variable elimination.*

Component elimination. A lifting problem raises two requirements for (*aux*, *comb*), calculating the expected output (defined by *orig*) and calculating the auxiliary values (defined by *aux*). Component elimination separates these two requirements into two subtasks, synthesizes two sub-results from these subtasks, and then merges these sub-results into a valid solution. In other words, some components of (*aux*, *comb*) are eliminated in the subtasks by applying this method.

As discussed in Section 2.4, a possible decomposition method here is to keep *aux* unchanged and decompose *comb* into two sub-programs for calculating the expected output and the auxiliary values, respectively. Although this method works well on the *sndmin* task, it introduces auxiliary values only for the original program and thus may fail in solving some complex tasks, such as the *mss* task, where extra auxiliary values are required for calculating auxiliary values.

Example 4.1. Suppose now the synthesis procedure in Section 2.4 is applied to the *mss* task. One can verify that the auxiliary program $aux\ xs \triangleq (mps\ xs, mts\ xs)$, which exactly provides auxiliary values required by *mss*, will be synthesized from the first subtask generated by decomposing *comb* (similar to Formula 2). Then, the second subtask (similar to Formula 3) will be unrealizable, because no combination function can calculate the *mps* of the whole list using only *mps*, *mts*, and *mss* of the two halves, as shown in Figure 11. Therefore, the synthesis will fail.

We solve this issue by decomposing *aux* as well and thus allowing new auxiliary values to be introduced in the second subtask. Because *aux* provides auxiliary values for not only the original program but also itself, its form can be assumed as $aux_1 \vartriangle aux_2$, where aux_1 specifies auxiliary values used for the original program, and aux_2 specifies auxiliary values used **only** for *aux*.

Example 4.2. In the *mss* example, aux_1 corresponds to $mps \vartriangle mts$, and aux_2 corresponds to *sum*. In the *sndmin* example (Section 2.1), aux_1 corresponds to *min*, and aux_2 corresponds to the dummy program *null* that provides no auxiliary value.

Component elimination in *AutoLifter* decomposes *comb* into $comb_1 \vartriangle comb_2$, decomposes *aux* into $aux_1 \vartriangle aux_2$, and synthesizes $(aux_1, comb_1)$ and $(aux_2, comb_2)$ in two sequential subtasks. The following are the details of this decomposition method.

As shall show later, component elimination would be recursively applied to its subtask. To unify the recursive applications, we introduce a generalized version of the lifting problem, as below, where the usage of *orig* on the right-hand side is replaced with a separate known program *aval*, representing the *available* inputs of *comb*. Program *aval* is set to *orig* in the first application of component elimination and may change to other programs in the subsequent applications.

$$(orig \vartriangle aux)\ (op\ (c, \overline{a})) = comb\ (c, (aval \vartriangle aux)^n\ \overline{a}) \tag{10}$$

This method decomposes a generalized lifting problem in three steps.

(1) Synthesize $(aux_1, comb_1)$ for calculating the original result, from the specification below.

$$orig\ (op\ (c, \overline{a})) = comb_1\ (c, (aval \vartriangle aux_1)^n\ \overline{a}) \tag{11}$$

This subtask asks for a combinator $comb_1$ to calculate the expected output and an auxiliary program aux_1 to provide necessary auxiliary values.

(2) Given the (partial) auxiliary program aux_1 found in the first subtask, synthesize a corresponding $(aux_2, comb_2)$ for calculating the auxiliary values, from the specification below.

$$aux\ (op\ (c, \overline{a})) = comb_2\ (c, (aval \vartriangle aux)^n\ \overline{a}), \textbf{where}\ aux \triangleq aux_1 \vartriangle aux_2 \tag{12}$$

This subtask asks for a combinator $comb_2$ to calculate the auxiliary values. In this procedure, some extra auxiliary values may be required (Example 4.2) and this subtask allows these auxiliary values to be introduced as aux_2.

(3) Construct $(aux, comb)$ as $(aux_1 \vartriangle aux_2, comb_1' \vartriangle comb_2)$, where $comb_1'$ is almost the synthesized $comb_1$ except the input format. Note that $comb_1$ takes $(c, (aval \vartriangle aux_1)^n\ \overline{a})$ as the input but $comb$ takes $(c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a})$ instead. To use $comb_1$ as a component in $comb$, $comb_1'$ is defined as $comb_1 \circ (id \times trans^n)$, where operator $\circ$ represents the function composition and function $trans$ adapts the input format, defined as $trans\ (a, (b, c)) \triangleq (a, b)$.

Example 4.3. After applying component elimination to the mss task (Formula 9), the specification of the first subtask is as follows.

$$mss\ (xs_L +\!\!+\ xs_R) = comb_1\ ((), (mss \vartriangle aux_1)^2\ (xs_L, xs_R)) \tag{13}$$

One can verify that components $a\,@\,mps$, $a\,@\,mts$ and $c\,@\,mss$ in Figure 13 form a valid solution here. After this solution is synthesized, the specification of the second subtask is as below.

$$aux\ (xs_L +\!\!+\ xs_R) = comb_2\ ((), (mss \vartriangle aux)^2\ (xs_L, xs_R)), \textbf{where}\ aux \triangleq (mps \vartriangle mts) \vartriangle aux_2$$

This subtask allows introducing new auxiliary values as aux_2 and thus is realizable: one can verify that components $a\,@\,sum$, $c\,@\,mps$, $c\,@\,mts$, and $c\,@\,sum$ form a valid solution to this subtask.

However, as a drawback of introducing aux_2, the second subtask (Formula 12) becomes relational and cannot be solved by *PolyGen*. An important observation here is that this subtask can be reagrded as a generalized lifting problem with parameters $(orig, op, aval)$ set to $(aux_1, op, aval \vartriangle aux_1)$. Therefore, it can be solved by applying component elimination recursively.

Variable elimination. Variable elimination decomposes the first subtask of component elimination (Formula 11) by synthesizing aux_1 and $comb_1$ from two subtasks, respectively. In other words, a variable (i.e., a program to be synthesized) is eliminated in the subtasks by applying this method. The decomposition is conducted in two steps.

(1) Generate a subtask for aux_1 and synthesize from it. In the original task generated by component elimination (Formula 11), a valid aux_1 must ensure the existence of a combination function for $comb_1$ to implement; in other words, for any two assignments of $(c, \overline{a})$, the inputs of $comb_1$ (i.e., $(c, (aval \vartriangle aux_1)^n\ \overline{a})$) must be different when the outputs (i.e., $orig \vartriangle (op\ (c, \overline{a}))$) differ, as shown below.

$$orig\ (op\ (c, \overline{a})) \neq orig\ (op\ (c', \overline{a'})) \rightarrow (c, (aval \vartriangle aux_1)^n\ \overline{a}) \neq (c', (aval \vartriangle aux_1)^n\ \overline{a'})$$

Algorithm 1: The decomposition system of *AutoLifter*.

Input: A lifting problem $\mathsf{LP}(orig, op)$.
Output: A solution $(aux, comb)$ to lifting problem $\mathsf{LP}(orig, op)$.

1 **Function** `VariableElimination`$(orig, op, aval)$:
2 $\quad subtask_1 \leftarrow$ the first subtask generated by variable elimination (Formula 14);
3 $\quad aux_1 \leftarrow$ `InductiveSynthesisForAux`$(subtask_1)$;
4 $\quad subtask_2 \leftarrow$ the second subtask corresponding to aux_1 (Formula 15);
5 $\quad$ **return** $(aux_1,$ `InductiveSynthesisForComb`$(subtask_2))$;
6 **Function** `ComponentElimination`$(orig, op, aval)$:
7 $\quad (aux_1, comb_1) \leftarrow$ `VariableElimination`$(orig, op, aval)$;
8 $\quad$ **if** $aux_1 = null$ **then return** $(null, comb_1 \vartriangle null)$;
9 $\quad (aux_2, comb_2) \leftarrow$ `ComponentElimination`$(aux_1, op, aval \vartriangle aux_1)$;
10 $\quad$ **return** $(aux, comb)$ constructed from $(aux_1, comb_1)$ and $(aux_2, comb_2)$;
11 **return** `ComponentElimination` $(orig, op, orig)$;

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.

$$\boxed{\left(aval^n\ \overline{a} = aval^n\ \overline{a'} \land orig\ (op\ (c, \overline{a})) \neq orig\ (op\ (c, \overline{a'}))\right) \rightarrow aux_1{}^n\ \overline{a} \neq aux_1{}^n\ \overline{a'}} \tag{14}$$

(2) Given the aux_1 found in the first subtask, generate a subtask by putting it into the specification for $(comb_1, aux_1)$ (Formula 11), as shown below, and synthesize a corresponding $comb_1$.

$$\boxed{orig\ (op\ (c, \overline{a})) = comb_1\ (c, (aval \vartriangle aux_1)^n\ \overline{a})} \tag{15}$$

Example 4.4. After applying variable elimination to the first subtask generated in the previous example (Formula 13), the specification of the first subtask is shown below.

$$mss^2\ (xs_L, xs_R) = mss^2\ (xs'_L, xs'_R) \land mss\ (xs_L +\!\!+\ xs_R) \neq mss\ (xs'_L +\!\!+\ xs'_R)$$
$$\rightarrow aux_1{}^2\ (xs_L, xs_R) \neq aux_1{}^2\ (xs'_L, xs'_R)$$

Decomposition System. The decomposition system of *AutoLifter* (Algorithm 1) decomposes a lifting problem by repeatedly applying component elimination and variable elimination.

(1) *AutoLifter* regards the input lifting problem as a generalized one (Line 11) and applies component elimination to it (Lines 6-10). The first subtask is further decomposed by variable elimination (Line 7), and the second one is solved by recursively applying component elimination (Line 9). The recursion terminates when no new auxiliary value is found (Line 8).
(2) *AutoLifter* 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 4.5. Figure 14 shows the full decomposition performed by *AutoLifter* to synthesize the programs in Figure 13 for the *mss* task, where nodes are subtasks (#1 denotes the original problem), arrows indicate task decomposition, tags within nodes indicate the sub-program synthesized from each subtask, and different line styles indicate different decomposition/subtask types. As we can see, the scale of the leaf subtasks (#3, #4, #7, #8, #11, and #12, each including at most 2 expressions) is greatly reduced compared to the original lifting problem (#1, including 7 expressions).

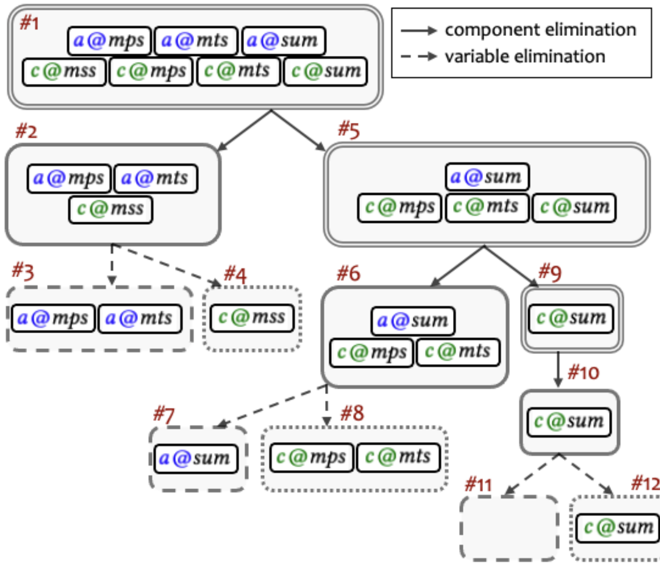

Fig. 14. The decomposition performed by *AutoLifter* to synthesize programs in Figure 13.

Table 3. The parameters of generalized lifting problems in Figure 14.

Subtask	*orig*	*aval*
#1	*sndmin*	*sndmin*
#5	$mps \vartriangle mts$	$sndmin \vartriangle (mps \vartriangle mts)$
#9	*sum*	$(sndmin \vartriangle (mps \vartriangle mts)) \vartriangle sum$

Table 4. Subtasks in Figure 14 solved by each function in Algorithm 1.

Function	Subtasks
`ComponentElimination`	#1, #5, #9
`VariableElimination`	#2, #6, #10
`InductiveSynthesisForAux`	#3, #7, #11
`InductiveSynthesisForComb`	#4, #8, #12

There are four types of subtasks: (1) the generalized lifting problem, 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 expressions in *aux*, and (4) the second subtasks of variable elimination (#4, #8, #12), which involve only expressions in *comb*. Table 3 lists the parameters of the three generalized lifting problems, and Table 4 lists the subtasks solved by each function in Algorithm 1.

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.

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 $(orig, op, aval, aux_1)$ specifying the task.
Output: A valid combinator $comb_1^*$.

1 $ioexamples \leftarrow \emptyset$;
2 **foreach** $(c, \overline{a}) \in examples$ **do**
3 $\quad input \leftarrow (c, (aval \vartriangle aux_1)^n\ \overline{a})$;
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 14 and 15), 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 paper, we assume the existence of the verifier for both types of leaf tasks and focus on the example-based synthesis tasks. In practice, the verifier can be selected among off-the-shelf ones on demand. For example, bounded model checking [Biere et al. 2003] can be used when all related programs are symbolically executable, and in our implementation, a probabilistic verifier based on random testing is used by default (Section 6).

Example-based synthesizer of *comb*. We begin with the leaf subtask of *comb* (Formula 15), 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 $(c, (aval \vartriangle aux_1)^n\ \overline{a})$. Therefore, those efficient synthesis algorithms relying on input-output examples are available for this task. The example-based synthesizer (Algorithm 3) just converts the given examples to input-output forms (Lines 2-6) and then passes them to existing synthesizers.

Example-based synthesizer of *aux*. An example generated from the leaf subtask of *aux* (Formula 14) is an assignment to $(c, \overline{a}, \overline{a'})$, and its constraint is ${aux_1}^n\ \overline{a} \neq {aux_1}^n\ \overline{a'}$. 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 happens only when the premise of the specification is true.

Example 4.6. Table 5 shows two examples possibly generated from the subtask in Example 4.4. 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 $mts \vartriangle mps$ satisfies both examples.

The example-based synthesizer of *aux* (Algorithm 4, denoted as $\mathcal{S}_{aux}$) is built upon *observational equivalence (OE)* [Udupa et al. 2013], an enumerative synthesizer (Line 10). OE enumerates programs from small to large in a bottom-up manner; in other words, it constructs larger programs by

Table 5. Two possible examples generated from the subtask in Example 4.4.

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

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 $involvedInputs \leftarrow \{a \mid (c, \overline{a}, \overline{a'}) \in examples \wedge (a \in \overline{a} \vee a \in \overline{a'})\}$;
2 $oe \leftarrow$ `ObservationalEquivalenceSolver`$(involvedInputs)$;
3 $\forall size \geq 0, programs[size] \leftarrow []$; $result \leftarrow \bot$;
4 **Function** `IsCovered`$(prog, size)$**:**
5   **return** $\exists size' \leq size, \exists prog' \in programs[size'], prog'$ satisfies all examples that are satisfied by *prog*;
6 **Function** `Insert`$(prog, size)$**:**
7   **if** *prog satisfies all examples* $\wedge result = \bot$ **then** $result \leftarrow prog$;
8   **if** $\neg$`IsCovered`$(prog, size)$ **then** $programs[size]$.`Append`$(prog)$;
9 **Function** `Extend()`**:**
10     $component \leftarrow oe$.`Next()`; `Insert`$(component, 1)$
11     $prePrograms \leftarrow programs$;
12   **foreach** $size \in [1, \dots, lim_c - 1]$ *and* $prog \in prePrograms[size]$ **do**
13   `Insert`$(prog \vartriangle component, size + 1)$;
14   **end**
15 `Insert`$(null, 0)$;
16 **while** $result = \bot$ **do** `Extend()`;
17 **return** *result*;

combining those existing smaller programs via language constructs. OE uses an effective pruning strategy to avoid maintaining multiple programs of identical input-output behaviors. This strategy is parameterized by an input set and prunes off those programs producing duplicated outputs on this input set (compared to existing programs).

Example 4.7. 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 since both programs output 2 from the only 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 inputs inside $\overline{a}$ and $\overline{a'}$. Therefore, OE can be applied by including all inputs involved in examples into the input set (Lines 1-2, Algorithm 4).

Example 4.8. When the example set is $\{\overline{xs}_1, \overline{xs}_2\}$ (examples in Example 4.6), 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. $\mathcal{S}_{aux}$ involves a specialized optimization to speed up synthesizing auxiliary programs. As an important observation, aux_1 often needs to generate multiple auxiliary values in practice, for example, *mps* and *mts* are both required for calculating *mss* in D&C. At this time, the form of aux_1 can be assumed as a tuple of components (i.e., $comp_1 \triangle \dots \triangle comp_k$), each in a smaller scale. To better synthesize such programs, we only invoke OE to generate basic components and combines these components explicitly on the top level (Lines 6-14).

$\mathcal{S}_{aux}$ maintains a program storage *programs* during the enumeration, where *programs*[*size*] stores existing programs formed by *size* components (Line 3). In each iteration, $\mathcal{S}_{aux}$ 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, $\mathcal{S}_{aux}$ is configured by an integer lim_c, representing the maximum number of components considered by the top-level combination (Line 12). Note that $\mathcal{S}_{aux}$ 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 speed up the combination, we propose an optimization method named *observational covering*, which shares the same idea of pruning larger programs that do not contribute more than smaller programs with respect to a set of pre-determined examples. Recall that an example $(c, \overline{a}, \overline{a'})$ here requires the auxiliary program to return different results on $\overline{a}$ and $\overline{a'}$. When the auxiliary program is formed as a tuple of components, it returns different results when any of its components return differently. Based on this observation, if there are two combined programs *prog* and *prog*′ satisfying (1) *prog* uses fewer components than *prog*′ and (2) *prog* satisfies all examples satisfied by *prog*′, the effect of *prog*′ must be covered by *prog* (in the sense of satisfying all examples using at most lim_c components) and thus *prog*′ can be skipped from the combination.

Example 4.9. When the example set is $\{\overline{xs}_1, \overline{xs}_2\}$ (Example 4.6), 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 $comp \triangle max$) satisfying both examples, there must exist another valid program $comp \triangle null$ (i.e., *comp*) using fewer components.

Using this property, $\mathcal{S}_{aux}$ skips all covered programs in combination. Only programs that are not covered by existing ones will be inserted into the storage for further combination (Lines 4-5 and 8).

Example 4.10. When the example set is $\{\overline{xs}_1, \overline{xs}_2\}$ (Example 4.6), the limit lim_c is 2, and the first three components returned by OE are *max*, *mps*, and *mts* in order, Algorithm 4 runs as follows.

- 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 keep 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.

4.4 Properties

Soundness. *AutoLifter* is sound when the verifiers of leaf subtasks are sound (Theorem 4.11). In every decomposition made by *AutoLifter*, the second subtask is always for completing the synthesis result of the first one into a valid solution for the original task (Section 4.2), which means, the result of each decomposition must be correct when both subtasks are solved correctly. Therefore, the soundness of *AutoLifter* is implied by that of inductive synthesizers for leaf subtasks, which depends only on the used verifiers.

THEOREM 4.11 (SOUNDNESS). *The result of AutoLifter (Algorithm 1) is valid for the original lifting problem if the verifiers of leaf subtasks accept only valid programs for respective subtasks.*

PROOF. Proofs of the lemmas and theorems in this paper are available in Appendix A.1. □

Incompleteness. In theory, *AutoLifter* is incomplete mainly because its decomposition system is incomplete. This system may decompose a realizable lifting problem into unrealizable subtasks. In detail, for both decomposition methods, the specification derived for the first subtask is not precise and cannot ensure the realizability of the subsequent second subtask.

- The first subtask of component elimination (Formula 11) only requires calculating the expected output (defined by *orig*). However, an aux_1 valid for this subtask may introduce auxiliary values that cannot be calculated using auxiliary programs in $\mathcal{L}_{aux}$ and combinators in $\mathcal{L}_{comb}$, making the second subtask (Formula 12) unrealizable.
- The first subtask of variable elimination (Formula 14) only requires aux_1 to provide enough auxiliary values such that a function exists for calculating the expected output. However, the expressiveness of $\mathcal{L}_{comb}$ may not be enough to implement such a function, making the second subtask (Formula 15) unrealizable.

Fortunately, such theoretical incompleteness of *AutoLifter* seldom exposes in practice. In our evaluation, *AutoLifter* can solve almost all realizable tasks within a short timeframe (Section 7). The direct reason for this phenomenon is the excellent practical performance of $\mathcal{S}_{aux}$, the synthesizer for the auxiliary program (Algorithm 4). Note that the decomposition procedure is fully determined by the result of $\mathcal{S}_{aux}$: the specification of each subtask depends only on the original lifting problem and the auxiliary programs synthesized previously. In our evaluation, $\mathcal{S}_{aux}$ can always find the intended auxiliary program, making all subtasks generated by decomposition realizable.

As discussed in the *sndmin* example (Section 2.4), we ascribe the effectiveness of $\mathcal{S}_{aux}$ to two reasons: (1) the *compressing property* in a practical lifting problem (i.e., both *orig* and programs in $\mathcal{L}_{aux}$ map a large input space to a small output space) that makes the specification of *aux* (Formula 14) strong enough to exclude most candidates, and (2) the *preference to simpler auxiliary programs* that helps avoid those unnecessarily complex solutions. In the remainder of this section, we shall provide formal results corresponding to these two factors.

Note that there is another source of incompleteness in *AutoLifter*, that is, the usage of CEGIS for solving leaf subtasks. In theory, the iteration of CEGIS may not terminate when both the program space and the example space are infinite because incorrect programs may exist for any finite set of examples. We believe such incompleteness is minor in practice given the extensive applications of CEGIS. Therefore, we make the assumption below and shall use it in our later discussion.

ASSUMPTION 4.12 (COMPLETENESS OF CEGIS). *The verifiers used in CEGIS are sound and complete (i.e., accept a program if and only if it is valid), and the CEGIS iteration always terminates on a realizable synthesis task when the example-based synthesizer is sound and complete.*

Probabilistic completeness under the compressing property. To estimate the incompleteness quantitatively, we construct a probabilistic model of lifting problems and analyze the probability for *AutoLifter* to be incomplete on a random lifting problem sampled from the model. We prove that this probability tends to be 0 when an assumption on the expressiveness of $\mathcal{L}_{comb}$ and the compressing property holds (Corollary 4.17).

In our probabilistic model (denoted as $\mathcal{M}$), we model the semantics of every program in the program space as a uniformly random function. The detailed construction of $\mathcal{M}$ is shown below.

- $\mathcal{M}$ is constructed on a set of given parameters, including (1) the types of *orig* and *op* and (2) the syntax and the type of each program in $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$. The only thing $\mathcal{M}$ does in generating a lifting problem is to assign random semantics to each program.
- For simplicity, we assume (1) programs in $\mathcal{L}_{aux}$ are formed as tuples of components (i.e., $comp_1 \vartriangle \dots comp_k$), 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*, each auxiliary value, and the complementary input of *op*, and (3) the numbers of different values in types A (the input type of *orig*) and V are finite[2], denoted as s_A and s_V, respectively.
- $\mathcal{M}$ generates a lifting problem by uniformly sampling the semantics of each program from all functions in the corresponding type. For example, the semantics of *orig* is uniformly drawn from functions mapping from A to V ($s_A^{s_V}$ possibilities in total).

On a random realizable lifting problem sampled from $\mathcal{M}$, the failure rate (i.e., the probability of incompleteness) of *AutoLifter* is bounded by the sizes of input/output domains (s_A and s_V) and the minimum size of valid programs (Theorem 4.13).

Theorem 4.13 (Probabilistic Completeness). *When Assumption 4.12 is assumed, for any size limit lim_s and a random lifting problem φ drawn from $\mathcal{M}$, the failure probability of AutoLifter under the condition that φ has a valid solution no larger than lim_s is bounded, as shown below.*

$$\Pr_{\varphi\sim\mathcal{M}}\left[\textit{AutoLifter fails on } \varphi \,\middle|\, \exists(aux, comb)\Big(size(aux, comb) \leq lim_s \wedge (aux, comb) \textit{ is valid for } \varphi\Big)\right]$$
$$\leq 2^w\left(s_V^{-s_V} + s_V^{w+1}\exp\left(-s_A^n\Big/s_V^w\right)\right) \textbf{\textit{for}}\ w \triangleq (lim_c+1)lim_s$$

where $size(aux, comb)$ represents the total size of aux and comb, and lim_c is the parameter of $\mathcal{S}_{aux}$ (Algorithm 4), representing the number of components considered in the top-level combination.

The upper bound provided by Theorem 4.13 can be further refined using two domain properties. The first one is related to the expressiveness of $\mathcal{L}_{comb}$. As we can see, this upper bound provided by Theorem 4.13 is formed by two separate terms (marked as blue and green, respectively) which correspond to two different cases making *AutoLifter* fail.

- The first term corresponds to the case where a combination function exists without any auxiliary value but cannot be implemented in $\mathcal{L}_{comb}$. For example, when applying D&C to calculate the sum of a list, a combination function $comb\ (sum_L, sum_R) = sum_L + sum_R$ exists directly, but some auxiliary values may be required if operator $+$ is unavailable in $\mathcal{L}_{comb}$. In this case, $\mathcal{S}_{aux}$ will always synthesize *null* (the empty auxiliary program) and thus leads to an unrealizable subtask for the combinator.
- The second term corresponds to the case where no combination function exists unless auxiliary values are introduced, and an incorrect auxiliary program is synthesized by $\mathcal{S}_{aux}$.

However, the first case seldom happens in practice. On the one hand, $\mathcal{L}_{comb}$ used in practice is usually expressive, for example, the one in our implementation (Section 6) can express complex scalar calculations via nested branch operators (i.e., *if-then-else*). On the other hand, intuitively, tasks where a combination function directly exists should be easier (compared to the others) so the implementation of their combination function should be simpler. Therefore, we consider an assumption that the expressiveness of $\mathcal{L}_{comb}$ is enough for these directly existing combination functions (Assumption 4.14) and refine the upper bound in Theorem 4.13 (Corollary 4.15).

[2]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).

Assumption 4.14 (Expressiveness of $\mathcal{L}_{COMB}$). *For any lifting problem, if a combination function exists without any auxiliary values, a corresponding combinator exists in* $\mathcal{L}_{comb}$.

Corollary 4.15. *When Assumption 4.14 is further assumed, the upper bound in Theorem 4.13 can be tightened to the expression below.*

$$2^w\left(s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right), \textbf{\textit{where}}\ w \triangleq (lim_c+1)\,lim_s$$

The second useful domain property is the *compressing property* of practical lifting problems, that is, the input domains of *orig* and auxiliary programs in $\mathcal{L}_{aux}$ are usually far larger than their output domains. Recall the meaning of a lifting problem in the optimization sense (Section 3.2). The inputs of these programs correspond to the intermediate data structures constructed in an inefficient program, and their outputs correspond to the values calculated after eliminating these data structures. To achieve optimization, these programs must summarize a small result (e.g., a scalar value) from a large input (e.g., a data structure that can be arbitrarily large), leading to the compressing property.

Example 4.16. In the two lifting problems discussed in Sections 2.1 and 3.1 (Formulas 1 and 7), 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 size limit of lists tends to ∞ and every integer is bounded within a fixed range.

The compressing property can be reflected as $s_A \gg s_V$ in our model $\mathcal{M}$, under which the refined upper bound in Corollary 4.15 further becomes negligible (Corollary 4.17).

Corollary 4.17. *The upper bound in Corollary 4.15 tends to* 0 *when* s_A/s_V^w *tend to* ∞.

Preference of $\mathcal{S}_{aux}$ for simpler programs. In the analysis above, we model the semantics of different programs as independent to ease the probabilistic 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 specification for *aux* (Formula 14) 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 4.18. In the *sndmin* example (Section 2.4), many valid programs for the *aux* subtask (Formula 5) 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 $\mathcal{L}_{comb}^{ex}$ (Figure 5b).

The key for *AutoLifter* to perform well under such dependency is its preference for simpler auxiliary programs, following the principle of *Occam's razor*. As an enumeration-based synthesizer, $\mathcal{S}_{aux}$ always returns a minimal program satisfying the leaf subtask (Theorem 4.19) and thus can successfully avoid those unnecessarily complex programs derived from the target one.

Theorem 4.19 (Minimality). *Given an example-based synthesis task for the auxiliary program, let* aux^* *be the synthesis result of* $\mathcal{S}_{aux}$. *Then, any strict sub-program of* aux^* *must not be valid for the given task.*

5 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 inductive synthesizers for the corresponding paradigms. In this section, we shall supply more details on how

```
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.

the efficiency of the resulting program is ensured in these reductions (Section 5.1) and provide reductions for several other paradigms (Section 5.2).

5.1 Efficiency Condition

For a synthesizer for applying algorithmic paradigms, it is crucial to ensure the efficiency of the resulting program. We require the domain-specific languages $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ to provide the efficiency guarantee when instantiating *AutoLifter*, following the SyGuS framework. These languages need to ensure that every possible solution can always result in an efficient enough program. Each time when instantiating *AutoLifter* for an algorithmic paradigm, different efficiency guarantees can be established by using different languages.

In this paper, we consider a condition (denoted as the *efficiency condition*) under which satisfactory efficiency guarantees can be established for many paradigms. 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 5.1. The domain-specific languages $\mathcal{L}_{aux}^{ex}$ and $\mathcal{L}_{comb}^{ex}$ (Figures 5a and 5b, discussed in the *sndmin* example) satisfy the efficiency condition.

Under the efficiency condition, the efficiency of the D&C and incremental programs synthesized by the corresponding instantiation of *AutoLifter* is ensured as follows.

- The D&C program (synthesized through the reduction in Section 2.1) is ensured to be $O(n/p)$ time in parallel on a list of length n and $p \leq n/\log n$ processors when the original program *orig* runs in constant time on a singleton list. In this program, each of *orig*, *aux*, and *comb* is invoked $O(n)$ times, and the former two are only invoked on singleton lists.
- The incremental program (synthesized through the reduction in Section 4.1) is ensured to be constant-time per change because only *comb* is invoked once after each change.

5.2 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 for each element. To apply single-pass to an original program *orig*, an auxiliary program *aux* and a combinator *comb* satisfying the formula below 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 possible change, an auxiliary program *aux* and a combinator *comb* satisfying the formula below 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 be $O(1)$ time per change because only *comb* is invoked once after each change.

Example 5.2. Based on the previous incrementalization task (Section 4.1), 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 $\{\text{“}front\text{”}, \text{“}back\text{”}\} \times \texttt{Int}$ and the corresponding *change* is as follows.

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

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 l-th to the r-th 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, i.e., 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.

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

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 5.2. A possible operator *op* of the resulting lifting problem is shown below, 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.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	$\text{IntConst} \mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid sum\ N_{\mathbb{L}} \mid len\ N_{\mathbb{L}} \mid head\ N_{\mathbb{L}}$
		$\mid$	$last\ N_{\mathbb{L}} \mid access\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid count\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid min\ N_{\mathbb{L}}$
		$\mid$	$max\ N_{\mathbb{L}} \mid neg\ N_{\mathbb{Z}}$
List expr	$N_{\mathbb{L}}$	$\rightarrow$	$\text{Input} \mid take\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid drop\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid rev\ N_{\mathbb{L}}$
		$\mid$	$map\ F_{\mathbb{Z}}\ N_{\mathbb{L}} \mid filter\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid zip \oplus N_{\mathbb{L}}\ N_{\mathbb{L}} \mid sort\ N_{\mathbb{L}}$
		$\mid$	$scanl \oplus N_{\mathbb{L}} \mid scanr \oplus N_{\mathbb{L}}$
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 $\mathcal{L}_{aux}$.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid (S, S)$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	$\text{IntConst} \mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid if\ N_{\mathbb{B}}\ then\ N_{\mathbb{Z}}\ else\ N_{\mathbb{Z}} \mid N_{\mathbb{T}}.i$
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	$N_{\mathbb{T}}$	$\rightarrow$	$\text{Input} \mid N_{\mathbb{T}}.i$
Binary Operator	$\oplus$	$\rightarrow$	$+ \mid - \mid \times \mid div$

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

Segment Trees. A segment tree is a classical data structure for efficiently answering queries about 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.

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

Through the construction in Example 5.2, 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.

6 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).

We use the language of *DeepCoder* [Balog et al. 2017] as $\mathcal{L}_{aux}$ in our implementation (Figure 17). 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 6.1. 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 shown below.

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

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

We use the language of the conditional arithmetic domain in SyGuS-Comp [Alur et al. 2019] as $\mathcal{L}_{comb}$ in our implementation. Similar to the case of $\mathcal{L}_{aux}$, we add operators for accessing and constructing tuples (i.e., $N_{\mathbb{T}}.i$ and (S, S), where the types of expressions expanded from $N_{\mathbb{T}}$ are tuples) to deal with the cases requiring multiple auxiliary values.

These languages match the assumptions we used when analyzing the completeness (Section 4.4, assuming the expressiveness of $\mathcal{L}_{comb}$ and the compressing property of L_{aux}) and the efficiency of the resulting program (Section 5.1, assuming the efficiency condition).

- For expressiveness, $\mathcal{L}_{comb}$ can express complex integer calculations via nested branch operators. Although it cannot provide the strict guarantee in Assumption 4.14, we believe it can handle those easy tasks in practice where a combination function directly exist.
- 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[3].
- 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 5.

Verification. *AutoLifter* applies the CEGIS framework to solve leaf subtasks and requires corresponding verifiers to generate counter-examples for incorrect programs (Section 4.3). A probabilistic verifier is used by default in our implementation. It generates random examples from a pre-defined distribution and then tests the candidate program on these examples. Such a verifier does not access the source code of *orig* and thus keeps the generality of *AutoLifter*.

This verifier can provide a probabilistic correctness guarantee for the CEGIS result when the number of tested examples is large enough. In our implementation, $10^4 \times i$ random examples are used in the i-th CEGIS iteration. Under this configuration, the probability for the error rate of the synthesized program (on a random example) to be more than 10^{-3} is at most 4.55×10^{-5}. The details of this guarantee can be found in Appendix A.2.

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.

[3]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 8.

- Every integer is uniformly drawn from integers in $[-5, 5]$ by default. This range will be changed for tasks having specialized requirements on the input list. For example, some tasks in the dataset collected by Farzan and Nicolet [2021b] consider only 01 lists, and thus the range of integers will be changed to $[0, 1]$ for them.

Other configurations. A synthesizer based on input-output examples is required to solve the leaf subtasks of *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, the maximum number of components in the top-level combination. We set lim_c to 4 in our implementation because 4 auxiliary values are already enough for most known lifting problems.

7 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*?

7.1 Experimental Setup

Baseline Solvers. Two general-purpose inductive synthesizers, *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018], are considered in our evaluation. Both of them can be applied to lifting problems and thus can be instantiated to synthesizers for various paradigms as *AutoLifter* does.

- *Enum* [Alur et al. 2013] is an enumerative solver. Given a lifting problem, *Enum* enumerates all possible (*aux*, *comb*) in the increasing order of the size 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.

Both *Enum* and *Relish* are re-implemented to support the list-related operators used in our paper.

Besides, *AutoLifter* is also compared with two state-of-the-art specialized synthesizers.

- *Parsynt* [Farzan and Nicolet 2017, 2021b] is a deductive synthesizer for D&C. It requires the original program to be single-pass, extracts *aux* directly by transforming the loop body with pre-defined rules, and synthesizes a corresponding *comb* via an inductive synthesizer. There are two versions of *Parsynt* available, denoted as *Parsynt17* [Farzan and Nicolet 2017] and *Parsynt21* [Farzan and Nicolet 2021b], where different transformation systems are used. Both of them are considered in our evaluation.
- *DPASyn* [Pu et al. 2011] is an inductive synthesizer for single-pass. It reduces the synthesis task of single-pass programs to a Sketch problem [Solar-Lezama 2013], solves it using existing Sketch solvers, and performs 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.

Dataset. Our evaluation is conducted on a dataset of 96 tasks of applying D&C-like algorithmic paradigms (Table 6). They are related to the four algorithmic problems discussed in Section 5.

Table 6. 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

- *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][4], including all tasks used by Farzan and Nicolet [2017] and Bird [1989a] and 12 out of 22 tasks used by Farzan and Nicolet [2021b]. The other 10 tasks used by Farzan and Nicolet [2021b] are out of the scope of *AutoLifter* because they cannot be reduced to lifting problems. They require a more general form of D&C where the divide operator is not determined, making our reduction inapplicable.
- *Problem 2: applying single-pass to a program.* We consider the tasks used in the evaluation of *DPASyn* [Pu et al. 2011] and include 4 out of 5 tasks into our dataset. The last task includes multiple input lists and is not supported by our current implementation.
  Besides, we construct a series of tasks from our D&C dataset as a supplement. For each D&C task, a single-pass task with the same original program is constructed[5]. These tasks are useful in removing the restriction on the input program from existing deductive synthesizers for D&C: one can first apply *AutoLifter* to get a single-pass program and then use deductive synthesizers to generate a D&C program.
- *Problem 3: Longest Segment Problem.* Zantema [1992] proposes three algorithmic paradigms for longest segment problems and discusses 3, 1, and 4 example tasks, respectively. For each example task, we include the task of applying the respective paradigm in our dataset.
- *Problem 4: applying segment trees to answer queries on a specific property of a segment in a list.* Because no previous work on segment trees provides a dataset, we search on Codeforces (https://codeforces.com/), a website for competitive programming, using keywords "segment tree" and "lazy propagation"[6]. We collect 13 tasks in this way and include them in our dataset.

Similar to 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], we assume the paradigm to be applied is given and directly apply the corresponding instantiation of *AutoLifter* to each task in our evaluation. In practice, when there are multiple paradigms available, we can either invoke all corresponding synthesizers in parallel or design a selector to select among them. Such a selection is out of the scope of this paper and is a direction for future work.

Configuration. Our experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor.

7.2 RQ1: Comparison of Synthesizers for Lifting Problems

Procedure. We compare *AutoLifter* with *Enum* and *Relish* on all tasks in our dataset with a time limit of 300 seconds and a memory limit of 8 GB. We record the time cost of each successful synthesis to measure the efficiency of the solvers.

Results. The results of this experiment are summarized in Table 7. For each solver, we report the number of solved tasks in column #Solved, its average time cost (seconds) on solved tasks in column $T_{\texttt{Base}}$, and the average time cost of *AutoLifter* on the same tasks in column $T_{\texttt{Ours}}$. We conduct the two manual analyses below on the synthesis results.

[4]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.

[5]A duplicated task involved by both *DPASyn* and our D&C dataset is ignored in this construction.

[6]A common alias of segment trees.

Table 7. The results of comparing *AutoLifter* with *Enum* and *Relish*.

Solver	D&C			Single-pass		
	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}
AutoLifter	**29**/36	10.4		**33**/39	1.74	
Enum	5/36	9.12	**0.06**	9/39	2.87	**0.10**
Relish	12/36	28.6	**6.68**	16/39	10.2	**2.11**

Solver	Longest Segment			Segment Tree			Total		
	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}
AutoLifter	**7**/8	2.26		**13**/13	12.3		**82**/96	6.53	
Enum	1/8	4.58	**0.14**	4/13	36.7	**0.31**	19/96	11.7	**0.14**
Relish	3/8	1.10	**0.36**	7/13	44.3	**12.5**	38/96	21.5	**5.33**

- We manually verify all results and confirm that they are all **completely correct**, though the verifier in our implementation provides only a probabilistic correctness guarantee[7].
- 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 (Corollary 4.17).

The results show that *AutoLifter* significantly outperforms the baseline solvers. It not only solves much more tasks but also solves much faster on those jointly solved tasks.

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

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

7.3 RQ2: Comparison with Synthesizers for Divide-and-Conquer

Procedure. We compare *AutoLifter* with the two versions of *Parsynt* on D&C tasks in our dataset and provide a single-pass implementation for each task to invoke *Parsynt*[8]. This comparison favors *Parsynt* because it can access those auxiliary values provided in the single-pass implementation.

We failed in installing *Parsynt17* because of some dependency issue, which is confirmed by the authors of *Parsynt17* but has not been solved yet. Therefore, we compare with *Parsynt17* only on its original dataset using the evaluation results reported by Farzan and Nicolet [2017].

Similar to the previous experiment, we use a time limit of 300 seconds and a memory limit of 8 GB and record the time cost of each successful synthesis. Please note that there is no difference between the time complexity of the synthesized programs because both *AutoLifter* and *Parsynt* ensure the time complexity to be exactly $\Theta(n/p)$ when a parallel template is used.

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

Table 8. The results of comparing *AutoLifter* with *Parsynt*.

Solver	#Tasks	$\#S_{\texttt{Base}}$	$\#S_{\texttt{Ours}}$	$T_{\texttt{Base}}$	$T_{\texttt{Ours}}$	$\#\text{Aux}_{\texttt{SP}}$
Parsynt17	20	**19**	**19**	15.6	**5.84**	39.3%
Parsynt21	36	24	**29**	6.86	**4.19**	56.9%

Results. The results of this experiment are summarized in Table 8. We report the number of tasks in each comparison in column #Tasks, the numbers of tasks solved by *Parsynt* and *AutoLifter* in columns $\#S_{\texttt{Base}}$ and $\#S_{\texttt{Ours}}$, the average time cost (seconds) of the two solvers in columns $\#T_{\texttt{Base}}$ and $\#T_{\texttt{Ours}}$, and the ratio of the number of auxiliary values in the provided single-pass program to the number of auxiliary values used in the D&C program synthesized by *Parsynt* in column $\#\text{Aux}_{\texttt{SP}}\%$. We consider only those tasks solved by both *Parsynt* and *AutoLifter* when calculating the average time cost and the ratio of provided auxiliary values.

The results show that *AutoLifter* offers competitive performance on synthesizing D&C programs compared to *Parsynt* even though *Parsynt* takes much more input, including 40%-60% of auxiliary values and the syntactic information.

Now, we would like to report two observations on the synthesis results.

(1) The result of *AutoLifter* never uses more auxiliary values than that of *Parsynt* and uses strictly fewer on 10 tasks. This is because the syntactic information may mislead *Parsynt* to unnecessarily complex solutions. For example, the original program of task *line_sight* (*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 directly 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 as it synthesizes directly from the semantics.

(2) When applying D&C, the issue of missing operators on *AutoLifter* (Section 7.2) can be alleviated by combining *AutoLifter* with *Parsynt*. Although the default languages are not expressive enough for applying D&C on 7 tasks, they are enough for applying single-pass on 5 tasks among these tasks. *AutoLifter* can successfully synthesize single-pass programs for these tasks, and then *Parsynt21* can synthesize D&C programs for 4 among them. In this way, the combination of *AutoLifter* and *Parsynt* can solve 33 out of 36 tasks, outperforming both individual solvers.

7.4 RQ3: Comparison with Synthesizers for Single-Pass

Procedure. We compare *AutoLifter* with *DPASyn* on all single-pass tasks in our dataset with a time limit of 300 seconds and a memory limit of 8 GB. The time cost of each successful synthesis is recorded to measure the efficiency of the solvers. Please note that there is no difference between the time complexity of the synthesized programs because both *AutoLifter* and *DPASyn* ensure the time complexity to be exactly $\Theta(n)$.

Besides, we also consider an enhanced configuration of *DPASyn* (denoted as $DPASyn_+$) in this experiment, where more compact program spaces are used. As a Sketch-based synthesizer, the time cost of *DPASyn* increases dramatically when the scale of the target program increases. However, the program space $\mathcal{L}_{comb}$ we used (Figure 18)[9] is so basic that a large program may be required for some simple functions. For example, the maximum of two integers $\max(a, b)$ has to be implemented as $ite(a \leq b, a, b)$ in $\mathcal{L}_{comb}$. Therefore, to better reveal the ability of *DPASyn*, we customize the program

[9] *DPASyn* does not synthesize the auxiliary program explicitly and thus never use the other program space $\mathcal{L}_{aux}$.

Table 9. The results of comparing *AutoLifter* with $AutoLifter_{\mathrm{OE}}$.

Solver	D&C			Single-pass		
	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}
AutoLifter	**29**/36	10.4		**33**/39	1.74	
$AutoLifter_{\mathrm{OE}}$	13/36	1.43	**0.33**	28/39	2.10	**1.44**

Solver	Longest Segment			Segment Tree			Total		
	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}	#Solved	T_{Base}	T_{Ours}
AutoLifter	**7**/8	2.26		**13**/13	12.3		**82**/96	6.53	
$AutoLifter_{\mathrm{OE}}$	6/8	8.48	**2.54**	8/13	14.2	**11.5**	55/96	4.40	**2.76**

space for each task when evaluating $DPASyn_+$, where (1) operators max and min are available, and (2) only those necessary operators are included. Note that the comparison between *AutoLifter* and $DPASyn_+$ favors the latter because $DPASyn_+$ explores a much smaller program space.

Results. The results of this experiment are summarized in the right-side table, organized similarly to the table of the first experiment (Table 7, Section 7.2). These results demonstrate that *AutoLifter* significantly outperforms both versions of *DPASyn* on both the number of solved tasks and the efficiency.

Solver	#Solved	T_{Base}	T_{Ours}
AutoLifter	**33**/39	1.74	
DPASyn	15/39	10.3	**2.23**
$DPASyn_+$	21/39	27.7	**1.75**

A major advantage of *AutoLifter* 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.

7.5 RQ4: Comparison with the Variant without Observational Covering

Procedure. *AutoLifter* involves a specialized optimization named observational covering when solving the leaf subtasks of *aux* (Section 4.3). To test the effect of this optimization, we consider a variant of *AutoLifter* where leaf subtasks of *aux* are directly solved by OE (denoted as $AutoLifter_{\mathrm{OE}}$) and compare it with the default *AutoLifter*. Similar to previous experiments, we use a time limit of 300 seconds and a memory limit of 8 GB and record the time cost of each successful synthesis.

Results. The results of this experiment are summarized in Table 9, organized similarly to the table of the first experiment (Table 7, Section 7.2). They show that observational covering significantly improves the efficiency of *AutoLifter*. Note that even when observational covering is removed, $AutoLifter_{\mathrm{OE}}$ still outperforms *Enum* and *Relish* on every problem (Table 7) and outperforms *DPASyn* (and $DPASyn_+$) on synthesizing single-pass programs (Table 9). This result also demonstrates the effectiveness of our decomposition system.

7.6 Case Study

We also conduct a case study on two tasks in our dataset, showing (1) the advantage of inductive synthesis and (2) the ability of *AutoLifter* on solving tasks difficult for human programmers.

Maximum segment product. The first task is named as *maximum segment product (msp)* [Bird 1989a], which is an advanced version of *mss* (Section 4.1). Given list $xs[1 \dots n]$, 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, by inductive synthesis, *AutoLifter* successfully solves this task using only 92.5 seconds 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 algorithm 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 players in competitive programming. It was set as a problem in 2020-2021 Winter Petrozavodsk Camp, a worldwide training camp representing the highest level of competitive programming. Only 26 out of 243 participating teams solved this problem within the 5-hour competition.

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

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

where *orig* represents a correct program for this longest segment problem. However, finding proper *aux* and *comb* is still difficult even after applying the paradigm. 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* within 10.2 seconds. The synthesized *comb* includes 152 AST nodes and is formed by several components dealing with different cases. Here, we only explain the component for calculating the expected output under the condition that $max\ xs_L \geq max\ xs_R$, as shown below.

$$comb\ (v, (res_L, res_R)) \triangleq \begin{cases} \max(lsp_R, \min(len_L + len_R + 1, v + max_L - 1)) & v + max\ xs_L > length\ xs_L + 1 \\ \max(lsp_L, lsp_R) & otherwise \end{cases}$$
$$\textbf{where}\ res_L \text{ is unfolded to } (lsp_L, (len_L, max_L)),\ res_R \text{ is unfolded to } (lsp_R, (len_R, max_R))$$
$$\textbf{assuming}\ max\ xs_L \geq max\ xs_R$$

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, and the length of s_v is $\min(len_L + len_R + 1, v + max_L - 1)$.

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 derivation below.

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

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, because 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 utilizes several tricky properties, and finding such a combinator is challenging for a human user. In contrast, *AutoLifter* can solve this problem quickly.

8 DISCUSSION

Dealing with incompleteness. The effectiveness of *AutoLifter* comes from its decomposition system. In this paper, we pay a lot of effort into arguing that the decomposition system should be effective in practice though it is incomplete in theory, and then use this incomplete system directly in our implementation. However, the latter design choice may seem unusual, especially when compared with existing deductive systems [Farzan and Nicolet 2017; Gulwani 2011; Huang et al. 2020; Polikarpova and Sergey 2019]. A more standard way is to build a scheduling mechanism above the decomposition system and explores all possible subtasks in order. For example, a backtracking mechanism can be used. Whenever an unrealizable subtask is generated, it goes back to previous leaf subtasks of *aux* (recall that the decomposition procedure is fully determined by the synthesized auxiliary programs) and switches to other possible auxiliary programs.

At a first glance, integrating the backtracking mechanism seems like a good choice. One may find that it can resolve the incompleteness of *AutoLifter* without any obvious loss of efficiency. As we argued before, the first choice of the decomposition seldom leads to unrealizable subtasks in practice. Therefore, the backtracking mechanism will seldom be activated if it is applied. In most cases, *AutoLifter* with backtracking just acts the same as our implementation.

Even so, the backtracking mechanism still faces two challenges below.

- First, the backtracking mechanism requires proving the unrealizability of synthesis tasks. Although some research progresses have been made [Hu et al. 2020; Kim et al. 2021], proving unrealizability is Turing-unrecognizable in theory and is still time-consuming in practice.
- Second, some efficiency synthesis techniques will be unavailable for synthesizing *aux* when achieving completeness by backtracking. Concretely, the synthesizer for *aux* needs to ensure that every possible auxiliary program will be considered after backtracking enough times, otherwise, the target solution may be missed. However, most synthesis techniques (e.g., observational equivalence and observational covering we used) fail in satisfying this

Table 10. 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 7.6)
Max Output	n	m		$n \times m$			n^m

requirement. They are designed only for synthesizing a single program and may skip many non-optimal programs during the synthesis for efficiency.

After considering the above factors comprehensively, we decide to use the incomplete decomposition system directly in our implementation. We believe the incompleteness introduced by this choice does not affect the effectiveness in practice, as demonstrated by our evaluation results.

The compressing property. In this paper, we discuss a lot about 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. Note that there is no strict limit on how much the input domain should be larger than the output domain. As shown in our probabilistic bound (Corollary 4.15), the probability for *AutoLifter* to be incomplete (on a random lifting problem sampled from our model) will be smaller on a larger input domain (compared to the output domain), and this probability finally converges to 0 when the size of the input domain approaches infinity.

Please note that the compressing property is never a sufficient condition for the completeness of *AutoLifter*. It is still possible to construct a realizable lifting problem on which this property holds but *AutoLifter* fails. We study the compressing property in this paper only for explaining the effectiveness of *AutoLifter* and clarifying the boundary of *AutoLifter*.

We regard those programs mapping from integer lists to a constant number of integers as compressing in previous discussions (Example 4.16 and Section 6). 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 $[-m, m]$. Non-compressing programs may exist if such enlargement is not limited, and in theory, injective functions from integer lists to a single integer exist when the range of the output integer is not bounded.

Luckily, the extreme case seldom happens in practice. Table 10 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 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) 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 (the default in our implementation, Section 6). Consequently, there are only $O(n^3)$ possible outputs of *msp* from 11^n different inputs, 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 $[1, 2, \dots, n]$. This observation suggests interesting future work of applying *AutoLifter* to those tasks where not only auxiliary values but also auxiliary data structures are required.

9 RELATED WORK

Automatic applications of D&C-like paradigms. *AutoLifter* can be instantiated to applying a broad class of paradigms and thus is related to previous studies on applying these 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 deductive and require that the input program is implemented as single-pass. Compared with them, *AutoLifter* is an inductive synthesizer that does not require single-pass implementations and can offer competitive performance compared to the previous state-of-the-art approach (Section 7.3).

Second, Acar et al. [2005] proposes a deductive approach for incrementalization. It constructs the auxiliary value as an execution trace of the original program and lets the combinator re-evaluate only those operations affected by the change. Consequently, the efficiency of its result greatly depends on how the original program is implemented: this approach can generate an efficient program only when the execution trace is affected little by the change. For example, in the incrementalization task for *sndmin* (Section 3.1), the result 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 inductive synthesizer named *DPASyn* for single-pass, which 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 demonstrates the effectiveness of *AutoLifter* (Section 7.4).

Fourth, there exist multiple synthesizers that do not support synthesizing the auxiliary program. 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. Compared with these approaches, *AutoLifter* not only supports more paradigms but can also automatically find necessary auxiliary values.

At last, in a lifting problem, we assume the operator *op* is given and thus focus on synthesizing the auxiliary program *aux* and the combinator *comb*. In this sense, there are two related studies on a more general task where the operator is to be synthesized as well [Farzan and Nicolet 2021b; Miltner et al. 2022]. Their approaches and ours are complementary because Farzan and Nicolet [2021b]'s approach requires a single-pass implementation and Miltner et al. [2022]'s approach does not support synthesizing the auxiliary program. A possible future direction is to combine these approaches with *AutoLifter*.

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 need to synthesize the whole program from the start, where scalability becomes an issue. 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.

The divide-and-conquer-style synthesis framework of *AutoLifter* is similar to *DraydSynth* [Huang et al. 2020], which synthesizes programs by (1) transforming the synthesis task into separate subtasks by pre-defined rule, and (2) solving each subtask by enumerative solvers. However, the rules used in *DryadSynth* are specialized for Boolean and arithmetic operators and thus cannot be used for lifting problems, where these operators do not occur in the specification.

AutoLifter is also related to *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018] as they are applicable to lifting problems. We compare *AutoLifter* with both of them in our evaluation, and the results demonstrate the better effectiveness of *AutoLifter*.

10 CONCLUSION

In this paper, we study the problem of applying D&C-like algorithmic paradigms from the aspect of inductive synthesis. We capture the application of various paradigms as a novel class of synthesis problems, namely lifting problems, and propose a synthesizer *AutoLifter* for them. To address the scalability challenge, we propose two decomposition methods, namely component elimination and variable elimination, to divide a lifting problem into simpler subtasks and derive specifications for different parts of the synthesis target.

Although *AutoLifter* does not ensure completeness, we demonstrate its effectiveness from both theoretical and practical perspectives. In theory, we prove that *AutoLifter* seldom fails on a random lifting problem when the language of combinators is expressive enough and the compressing property holds. In practice, we evaluate *AutoLifter* on a dataset of 96 lifting problems and the results show that *AutoLifter* can solve most of these problems within a short timeframe.

Besides, we believe many techniques proposed in this paper are general and can be potentially applied to other tasks. For example, the method of 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. 2023].

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 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1205–1220. https://doi.org/10.1145/3183713.3196891

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 *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 1–8. http://ieeexplore.ieee.org/document/6679385/

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. *CoRR* abs/1904.07146 (2019). arXiv:1904.07146 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 *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* https://openreview.net/forum?id=ByldLrqlx

J. L. Bentley. 1977a. Solution to Klee"s Rectangle Problem. (1977).

Jon Louis Bentley. 1977b. Solutions to KleeâĂŹs rectangle problems. *Unpublished manuscript* (1977), 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 *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, 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 2017, Barcelona, Spain, June 18-23, 2017*. 540–555. https://doi.org/10.1145/3062341.3062355

Azadeh Farzan and Victor Nicolet. 2021a. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). 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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, 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 2017, Barcelona, Spain, June 18-23, 2017*, 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, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

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 2020, London, UK, June 15-20, 2020*, 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. *CoRR* abs/2103.04188 (2021). arXiv:2103.04188 https://arxiv.org/abs/2103.04188

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 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1159–1174. https://doi.org/10.1145/3385412.3386027

Susmit Jha and Sanjit A. Seshia. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* 54, 7 (2017), 693–726. https://doi.org/10.1007/s00236-017-0294-5

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. 2023. *Supplementary Material.* https://github.com/jiry17/AutoLifter

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 2019, Phoenix, AZ, USA, June 22-26, 2019*. 253–268. https://doi.org/10.1145/3314221.3314602

Joshua Lau and Angus Ritossa. 2021. Algorithms and Hardness for Multidimensional Range Updates and Queries. In *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference (LIPIcs, Vol. 185)*, James R. Lee (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:20. https://doi.org/10.4230/LIPIcs.ITCS.2021.35

Shu Lin, Na Meng, and Wenxin Li. 2019. Optimizing Constraint Solving via Dynamic Programming. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 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 Heidelberg, Berlin, Heidelberg. 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 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 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, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

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 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30. https://doi.org/10.1145/3290385

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 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 83–98. https://doi.org/10.1145/2048066.2048076

Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, 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 2015, Monterey, CA, USA, October 4-7, 2015*, 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 2017, Buenos Aires, Argentina, May 20-28, 2017*, 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 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 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 2006, San Jose, CA, USA, October 21-25, 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, Seattle, WA, USA, June 16-19, 2013*, 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. *PACMPL* 2, OOPSLA (2018), 155:1–155:27.

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

A APPENDIX: PROOFS AND GUARANTEES

This section provides the proofs for the theorems in this paper (Section 4.4) and supplies the details on the probabilistic correctness guarantee provided by our verifier (Section 6).

A.1 Proofs for Theorems

THEOREM A.1 (THEOREM 4.11). *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. It is enough to prove both decomposition methods are sound (i.e., the merged result is valid for the original task when the sub-results are valid for the corresponding subtasks) because the sub-programs synthesized from leaf subtasks must be valid when the verifiers are sound.

The case of variable elimination is straightforward, and for component elimination, we need to prove that for any $(aux_1, comb_1)$ and $(aux_2, comb_2)$ satisfying the formulas below, their combination $(aux_1 \vartriangle aux_2, comb_1' \vartriangle comb_2)$ must be valid for the corresponding (generalized) lifting problem.

$$orig\ (op\ (c, \overline{a})) = comb_1\ (c, (aval \vartriangle aux_1)^n\ \overline{a})$$
$$(aux_1 \vartriangle aux_2)\ (op\ (c, \overline{a})) = comb_2\ (c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a})$$

The correctness of this combination is implied by the derivation below.

$$\begin{aligned}
&(comb_1' \vartriangle comb_2')\ (c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a}) \\
=& \Big((comb_1 \circ (id \times trans_1^n))\ (c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a}), \\
&\qquad comb_2\ (c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a})\Big) \\
=& \Big(comb_1\ (c, (aval \vartriangle aux_1)^n\ \overline{a}), comb_2\ (c, (aval \vartriangle (aux_1 \vartriangle aux_2))^n\ \overline{a})\Big) \\
=& \Big(orig\ (op\ (c, \overline{a})), (aux_1 \vartriangle aux_2)\ (op\ (c, \overline{a}))\Big) \\
=& (orig \vartriangle (aux_1 \vartriangle aux_2))\ (op\ (c, \overline{a}))
\end{aligned}$$

where the first and the last formulas are the right-hand and left-hand sides of the specification of a (generalized) lifting problem, respectively. □

THEOREM A.2 (THEOREM 4.13). *When Assumption 4.12 is assumed, for any size limit lim_s and a random lifting problem φ drawn from $\mathcal{M}$, the failure probability of AutoLifter under the condition that φ has a valid solution no larger than lim_s is bounded, as shown below.*

$$\Pr_{\varphi \sim \mathcal{M}}\Big[\textit{AutoLifter fails on } \varphi\ \Big|\ \exists (aux, comb)\Big(size(aux, comb) \leq lim_s \wedge (aux, comb) \textit{ is valid for } \varphi\Big)\Big]$$
$$\leq 2^w\left(s_V^{-s_V} + s_V^{w+1}\exp\left(-s_A^n \big/ s_V^w\right)\right) \textbf{\textit{for }} w \triangleq (lim_c + 1) lim_s$$

where $size(aux, comb)$ represents the total size of aux and comb, and lim_c is the parameter of $\mathbb{S}_{aux}$ (Algorithm 4), representing the number of components considered in the top-level combination.

PROOF. For simplicity, we shall interchangeably use a synthesis task as a predicate, where $\varphi(prog)$ represents that *prog* is a valid program for task φ. Let $\tilde{\varphi}$ be the first *aux* subtask generated from φ, of which the specification is shown below.

$$(orig \vartriangle aux_1)^n\ \overline{a} = (orig \vartriangle aux_1)^n\ \overline{a'} \to orig\ (op\ (c, \overline{a})) = orig\ (op\ (c, \overline{a'}))$$

Besides, 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. With this notation, the condition of the target probability can be restated as $|\mathbb{A}(\varphi)| > 0$.

$$\mathbb{A}(\varphi) \triangleq \{aux \mid \exists comb, \varphi(aux, comb) \wedge \mathrm{size}(aux, comb) \leq lim_s\}$$

Step 1: a sufficient condition. Any program in $\mathbb{A}(\varphi)$ is valid for $\tilde{\varphi}$ by definition. Furthermore, under the assumption that CEGIS is complete (Assumption 4.12) and the verifiers are sound and complete, *AutoLifter* must not fail when a program in $\mathbb{A}(\varphi)$ is synthesized from $\tilde{\varphi}$. At this time, only 3 subtasks will be generated after $\tilde{\varphi}$, as shown below.

(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 program synthesized from $\tilde{\varphi}$. 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 a program in $\mathbb{A}(\varphi)$ is synthesized from $\tilde{\varphi}$ is a sufficient condition for the success of *AutoLifter*, and the target probability is bounded by the probability below.

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \,\middle|\, |\mathbb{A}(\varphi)| > 0\right] \tag{16}$$

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 is quite 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 in Formula 16 can be transformed as follows.

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \,\middle|\, |\mathbb{A}(\varphi)| > 0\right] \\
=\ &\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \wedge |\mathbb{A}(\varphi)| > 0\right] \Big/ \Pr_{\varphi\sim\mathcal{M}}\left[|\mathbb{A}(\varphi)| > 0\right] \\
=\ &\sum_{P\subseteq\mathcal{L}_{aux}} [|P|>0] \Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \notin P \wedge \mathbb{A}(\varphi) = P\right] \Big/ \sum_{P\subseteq\mathcal{L}_{aux}} [|P|>0] \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right] && (17) \\
\leq\ &\frac{\sum_{P\subseteq\mathcal{L}_{aux}} [|P|>0] \left(\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \notin P \wedge \mathbb{A}(\varphi) = P\right] + (|P|-1)\Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right]\right)}{\sum_{P\subseteq\mathcal{L}_{aux}} |P| \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right]} && (18)
\end{aligned}$$

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

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

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

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

- 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 probability 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 18, we further perform the derivation below, where $\mathbb{L}_{\leq lim_s}$ denotes the subspace of $\mathcal{L}_{aux}$ including only those programs of which the size is no larger than lim_s.

$$\begin{aligned}
\text{Formula 18} &= \sum_{P \subseteq \mathcal{L}_{aux}} \sum_{aux^* \in P} \Pr_{\varphi \in \mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \land \mathbb{A}(\varphi) = P\right] \Big/ \sum_{P \subseteq \mathcal{L}_{aux}} \sum_{aux^* \in P} \Pr_{\varphi \sim \mathcal{M}}\left[\mathbb{A}(\varphi) = P\right] \\
&= \sum_{aux^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \land aux^* \in \mathbb{A}(\varphi)\right] \Big/ \sum_{aux^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}}\left[aux^* \in \mathbb{A}(\varphi)\right] \\
&\leq \max_{aux^* \in \mathbb{L}_{\leq lim_s}} \left(\Pr_{\varphi \sim \mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \land aux^* \in \mathbb{A}(\varphi)\right] \Big/ \Pr_{\varphi \sim \mathcal{M}}\left[aux^* \in \mathbb{A}(\varphi)\right] \right) &\quad (19) \\
&= \max_{aux^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] &\quad (20)
\end{aligned}$$

Formula 19 uses the inequality below (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 16) 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 $\mathcal{S}_{aux}(\tilde{\varphi}) = aux^*$, and the conditional probability in Formula 20 can be bounded as follows.

$$\begin{aligned}
&\Pr_{\varphi \sim \mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] \\
\leq\ &\Pr_{\varphi \sim \mathcal{M}}\left[\exists aux \in \mathbb{L}_{\leq lim_c\, lim_s}, aux \neq aux^* \land \tilde{\varphi}(aux) \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] \\
\leq\ &\sum_{aux \in \mathbb{L}_{\leq lim_c\, lim_s}} [aux \neq aux^*] \Pr_{\varphi \sim \mathcal{M}}\left[\tilde{\varphi}(aux) \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] \\
\leq\ &2^{lim_s\, lim_c} \max_{aux \in \mathbb{L}_{\leq lim_c\, lim_s}} [aux \neq aux^*] \Pr_{\varphi \sim \mathcal{M}}\left[\tilde{\varphi}(aux) \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] &\quad (21)
\end{aligned}$$

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 right-side $\mathbb{A}(\varphi)$. The probability in Formula 21 can be transformed as follows, where $\mathbb{L}$ denotes the subset of $\mathcal{L}_{comb}$ including programs whose size is at most $lim_s - \text{size}(aux^*)$.

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \;\middle|\; aux^* \in \mathbb{A}(\varphi)\right] \\
&= \Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \land aux^* \in \mathbb{A}(\varphi)\right]\Big/ \Pr_{\varphi\sim\mathcal{M}}\left[aux^* \in \mathbb{A}(\varphi)\right] \\
&= \Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \land \exists comb \in \mathbb{L}, \varphi(aux^*, comb)\right]\Big/ \Pr_{\varphi\sim\mathcal{M}}\left[\exists comb \in \mathbb{L}, \varphi(aux^*, comb)\right] \\
&\le \sum_{comb\in\mathbb{L}} \Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \land \varphi(aux^*, comb)\right]\Big/ \max_{comb\in\mathbb{L}} \Pr_{\varphi\sim\mathcal{M}}\left[\varphi(aux^*, comb)\right] \\
&\le 2^{lim_s} \max_{comb\in\mathbb{L}} \Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \land \varphi(aux^*, comb)\right]\Big/ \max_{comb\in\mathbb{L}} \Pr_{\varphi\sim\mathcal{M}}\left[\varphi(aux^*, comb)\right]
\end{aligned} \tag{22}$$

The last step uses the fact that $|\mathbb{L}| \le 2^{lim_s}$, which is because programs in $\mathbb{L}$ are no larger than lim_s. So far, the right-side $\mathbb{A}(\varphi)$ in the original sufficient condition (Formula 16) has been eliminated.

Note that all well-typed combinators *comb* of aux^* are independent random functions drawn from the same set. Therefore, different choices of *comb* would not affect the probability, and the two max operators in Formula 22 can be safely ignored, as shown below, where *comb* denotes an arbitrary well-typed combinator of aux^*.

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \land \varphi(aux^*, comb)\right]\Big/ \Pr_{\varphi\sim\mathcal{M}}\left[\varphi(aux^*, comb)\right] \tag{23}$$

To bound this probability, we need to (1) find a lower bound for its denominator, and (2) find an upper bound for its nominator. In the remainder of our analysis, we regard $orig \circ op$ and $aux^* \circ op$ as individual random functions for simplicity and abbreviate them as *oo* and *ao*, respectively. Besides, we denote the size of the output domain of aux^* as s^* and denote $s^* \times s_V$ (the size of the output domain of $orig \vartriangle aux^*$) as s. Note that s^* is no larger than $s_V^{lim_s}$ because aux^* (whose size is no larger than lim_s) cannot return more than lim_s auxiliary values.

Step 5: Lower bound of the denominator. After unfolding the definition of φ, the denominator of Formula 23 becomes as follows.

$$\Pr_{\varphi\sim\mathcal{M}}\left[(oo \vartriangle ao)\ (c, \overline{a}) = comb\ (c, (orig \vartriangle aux^*)^n\ \overline{a})\right]$$

For every concrete assignment of $(comb, orig, aux^*)$, the right-hand side of the equation uniquely defines a function from $B \times A^n$ to the output type of $orig \vartriangle aux^*$. The probability for $oo \vartriangle ao$ to be exactly this function is equal to $\text{pow}(s, -s_V \cdot s_A^n)$ whatever the assignment of $(comb, orig, aux^*)$ is. Therefore, the denominator is equal to $\text{pow}(s, -s_V \cdot s_A^n)$.

Step 6: Upper bound of the nominator. After unfolding the definitions of φ and $\tilde{\varphi}$, the nominator of Formula 23 becomes as follows.

$$\begin{aligned}
\Pr_{\varphi\sim\mathcal{M}}\Big[&(oo \vartriangle ao)\ (c, \overline{a}) = comb\ (c, (orig \vartriangle aux^*)^n\ \overline{a}) \\
&\land\ (orig \vartriangle aux)^n\ \overline{a} = (orig \vartriangle aux)^n\ \overline{a} \to oo\ (c, \overline{a}) = oo\ (c, \overline{a'})\Big]
\end{aligned}$$

Let $\mathcal{E}$ be the event in this probability and let $\mathcal{E}_{sur}$ be the event that $orig \vartriangle aux \vartriangle aux^*$ is a surjection. The above probability can be decomposed into two parts, as shown below, and the remaining task is to derive upper bounds for both parts.

$$\Pr[\mathcal{E}] = \Pr[\mathcal{E} \mid \mathcal{E}_{sur}]\Pr[\mathcal{E}_{sur}] + \Pr[\mathcal{E} \mid \neg\mathcal{E}_{sur}]\Pr[\neg\mathcal{E}_{sur}] \le \Pr[\mathcal{E} \mid \mathcal{E}_{sur}] + \Pr[\mathcal{E} \mid \neg\mathcal{E}_{sur}]\Pr[\neg\mathcal{E}_{sur}]$$

Step 6.1: Upper bound of $\Pr[\mathcal{E} \mid \mathcal{E}_{sur}]$. Suppose the assignment to $(oo, ao, orig, aux, aux^*)$ satisfying $\tilde{\varphi}(aux)$ and $\mathcal{E}_{sur}$ is given. Under this assignment, $\Pr[\mathcal{E} \mid \mathcal{E}_{sur}]$ is equal to $\Pr[\varphi(aux^*, comb)]$. Because *comb* must output the same on the same input, a valid *comb* exists only when the following event (denoted as $\mathcal{E}_{comb}$) happens.

$$(orig \vartriangle aux^*)^n\ \overline{a} = (orig \vartriangle aux^*)^n\ \overline{a'} \rightarrow (oo \vartriangle ao)\ (c, \overline{a}) = (oo \vartriangle ao)\ (c, \overline{a'})$$

When $\mathcal{E}_{comb}$ happens, the valid *comb* has been uniquely determined by the assignment to the other functions. Because the sizes of the input and output domains of *comb* are s and $s_V \cdot s^n$, respectively, the probability of $\mathcal{E}$ is $\mathrm{pow}(s, -s_V \cdot s^n)$ at this time.

Now let us go back to the normal case where the assignment is not given. The analysis above suggests the following equation about $\Pr[\mathcal{E} \mid \mathcal{E}_{sur}]$.

$$\Pr[\mathcal{E} \mid \mathcal{E}_{sur}] = \mathrm{pow}(s, -s_V \cdot s^n) \Pr[\mathcal{E}_{comb} \wedge \tilde{\varphi}(aux) \mid \mathcal{E}_{sur}] \tag{24}$$

We claim that $\mathcal{E}_{comb} \wedge \tilde{\varphi}(aux)$ implies the event below (denoted as $\mathcal{E}_{imp}$) when $\mathcal{E}_{sur}$ happens.

$$\begin{aligned}&\left(orig^n\ \overline{a} = orig^n\ \overline{a'} \rightarrow oo\ (c, \overline{a}) = oo\ (c, \overline{a'})\right)\\ &\qquad \wedge\ \left((orig \vartriangle aux^*)^n\ \overline{a} = (orig \vartriangle aux^*)^n\ \overline{a'} \rightarrow ao\ (c, \overline{a}) = ao\ (c, \overline{a'})\right)\end{aligned}$$

To prove this claim, we need only to consider the first implication since the second one is direct from $\mathcal{E}_{oo}$. Consider any assignment to $(c, \overline{a}, \overline{a'})$ satisfying $orig^n\ \overline{a} = orig^n\ \overline{a'}$. Because $orig \vartriangle aux \vartriangle aux^*$ is a surjection, there exists $\overline{a''}$ in A^n such that (1) $orig^n\ \overline{a} = orig^n\ \overline{a'} = orig^n\ \overline{a''}$, (2) $aux^n\ \overline{a} = aux^n\ \overline{a''}$, and (3) $(aux^*)^n\ \overline{a'} = (aux^*)^n\ \overline{a''}$. At this time, $\tilde{\varphi}(aux)$ implies $oo\ (c, \overline{a}) = oo\ (c, \overline{a''})$, and $\mathcal{E}_{comb}$ implies that $oo\ (c, \overline{a'}) = oo\ (c, \overline{a''})$. Consequently, $oo\ (c, \overline{a}) = oo\ (c, \overline{a'})$ and the claim is proved.

The inequality below is obtained after applying the claim to Formula 24.

$$\Pr[\mathcal{E} \mid \mathcal{E}_{sur}] \leq \mathrm{pow}(s, -s_V \cdot s^n) \Pr[\mathcal{E}_{imp} \mid \mathcal{E}_{sur}]$$

The task remaining here is to calculate $\Pr[\mathcal{E}_{imp} \mid \mathcal{E}_{sur}]$. Suppose an assignment to $(orig, aux, aux^*)$ satisfying $\mathcal{E}_{sur}$ is given.

- For *oo*, its input domain is divided into exactly s_V^{n+1} groups according to $(c, orig^n\ \overline{a})$, and a valid *oo* is required to output the same in each group. There are $\mathrm{pow}(s_V, s_V^{n+1})$ functions satisfying this condition, so the probability for *oo* to be valid is $\mathrm{pow}(s_V, s_V^{n+1} - s_V \cdot s_A^n)$.
- Similarly, for *ao*, its input domain is divided into exactly $s_V \cdot s^n$ groups, and it is required to output the same in each group. So the probability for *ao* to be valid is $\mathrm{pow}(s^*, s_V \cdot s^n - s_V \cdot s_A^n)$.

The above two probabilities are independent with the concrete assignment to $(orig, aux, aux^*)$, so the equation below holds.

$$\begin{aligned}\Pr[\mathcal{E}_{imp} \mid \mathcal{E}_{sur}] &\leq \mathrm{pow}(s, -s_V \cdot s^n)\mathrm{pow}(s_V, s_V^{n+1} - s_V \cdot s_A^n)\mathrm{pow}(s^*, s_V \cdot s^n - s_V \cdot s_A^n)\\ &= \mathrm{pow}(s, -s_V \cdot s_A^n)\mathrm{pow}(s_V, s_V(s_V^n - s^n))\\ &\leq \mathrm{pow}(s, -s_V \cdot s_A^n)\mathrm{pow}(s_V, -s_V)\end{aligned}$$

Step 6.2: Upper bound of $\Pr[\neg\mathcal{E}_{sur}]$. The size of the input domain of $orig \vartriangle aux^* \vartriangle aux$ (abbreviated as $orig'$) is $s_V \cdot s_A^n$, and let s_o be the size of its output domain. Because the sizes of aux^* are at most lim_s and $lim_c lim_s$, respectively, the output include at most $(1 + lim_s + lim_c lim_s)$ values, so s_o is no larger than $\mathrm{pow}(s_V, 1 + lim_s + lim_c lim_s)$.

An upper bound of $\Pr[\neg\mathcal{E}_{sur}]$ can be obtained through the derivation below, where $\mathbb{I}$ and $\mathbb{O}$ denote the input and output domains of *orig'*.

$$\begin{aligned}\Pr[\neg\mathcal{E}_{sur}] &= \Pr\left[\exists o \in \mathbb{O}, \forall i \in \mathbb{I}, orig'\ i \neq o\right]\\ &\leq \sum_{o\in\mathbb{O}} \Pr[\forall i \in \mathbb{I}, orig'\ i \neq o]\\ &= s_o\,(1-1/s_o)^{s_V\cdot s_A^n}\\ &\leq s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right), \textbf{where } w \triangleq lim_s + lim_s\, lim_c\end{aligned}$$

Step 6.3: Upper bound of $\Pr[\mathcal{E} \mid \neg\mathcal{E}_{sur}]$. By the definition of $\mathcal{E}$, this probability is no larger than $\Pr[\varphi(aux^*, comb) \mid \mathcal{E}_{sur}]$. Note that event $\mathcal{E}_{sur}$ depends only on *orig*, *aux*, and aux^*, and Step 5 shows that $\Pr[\varphi(aux^*, comb)]$ is always $\mathrm{pow}(s, -s_V \cdot s_A^n)$ whatever $(orig, aux, aux^*)$ is assigned to. Therefore, the inequality below holds.

$$\Pr[\mathcal{E} \mid \neg\mathcal{E}_{sur}] \leq \Pr[\varphi(aux^*, comb) \mid \mathcal{E}_{sur}] = \mathrm{pow}(s, -s_V \cdot s_A^n)$$

Step 7: Summary. First, the inequality below is obtained by combining Steps 6, 6.1, 6.2, and 6.3.

$$\begin{aligned}\Pr[\mathcal{E}] &\leq \Pr[\mathcal{E} \mid \mathcal{E}_{sur}] + \Pr[\mathcal{E} \mid \neg\mathcal{E}_{sur}]\Pr[\neg\mathcal{E}_{sur}]\\ &\leq \mathrm{pow}(s, -s_V\cdot s_A^n)\left(\mathrm{pow}(s_V, -s_V) + s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right)\end{aligned}$$

Second, the inequality below is obtained by further combining Steps 4 and 5.

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(aux) \,\middle|\, aux^* \in \mathbb{A}(\varphi)\right] \leq 2^{lim_s}\left(\mathrm{pow}(s_V, -s_V) + s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right)$$

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

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{aux}(\tilde{\varphi}) \neq aux^* \,\middle|\, aux^* \in \mathbb{A}(\varphi)\right] \leq 2^{w}\left(\mathrm{pow}(s_V, -s_V) + s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right)$$

At last, we know the target probability is bounded by the formula below after further combining Steps 1 and 2, which is exactly the target theorem.

$$2^w\left(s_V^{-s_V} + s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right),\ \textbf{where } w \triangleq (lim_c+1)lim_s$$

□

Corollary A.3 (Corollary 4.15). *When Assumption 4.14 is further assumed, the upper bound in Theorem 4.13 can be tightened to the expression below.*

$$2^w\left(s_V^{w+1}\exp\left(-s_A^n\big/s_V^w\right)\right), \textbf{\textit{where }} w \triangleq (lim_c+1)lim_s$$

Proof. The proof is almost the same as that of Theorem 4.13. The difference happens in Step 6.1 when analyzing $\Pr[\mathcal{E} \mid \mathcal{E}_{sur}]$. In this case, a combination function directly exists without any auxiliary values (i.e., the empty auxiliary program *null* is valid for subtask $\tilde{\varphi}$), as shown below.

$$\begin{aligned}(orig \vartriangle null)^n\ \overline{a} = (orig \vartriangle null)^n\ \overline{a'} &\to orig\ (op\ (c, \overline{a})) = orig\ (op\ (c, \overline{a'}))\\ \iff orig^n\ \overline{a} = orig^n\ \overline{a'} &\to orig\ (op\ (c, \overline{a})) = orig\ (op\ (c, \overline{a'}))\end{aligned}$$

where the latter formula is implied by $\mathcal{E} \wedge \mathcal{E}_{sur}$, according to the proof of Theorem 4.13.

By Assumption 4.14, *AutoLifter* never fails in this case, so the corresponding probability will not contribute to the error rate of *AutoLifter*. Consequently, the first term in the bound of Theorem 4.13 is eliminated, and the target corollary is obtained. □

Theorem A.4 (Theorem 4.19). *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. Because 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^*. Because aux^* is formed as a tuple of components (i.e., in the form of $comp_1 \vartriangle \dots \vartriangle comp_k$), 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 Probabilistic Correctness Guarantee of Our Verifier

Our implementation uses a probabilistic verifier based on random sampling (Section 6). In the i-th CEGIS iteration, it tests the candidate program using $10^4 \times i$ random examples generated from a pre-defined distribution. As we claimed before, this verifier can provide a probabilistic 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}. Now we provide the details of this guarantee.

To begin with, let us 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 our 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 i-th 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.

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

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 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 5.1), i.e., the efficiency guarantee provided by *AutoLifter* in our implementation.

B.1 The first algorithmic paradigm for the longest segment problem

The 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*, which is defined as the following.

```
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.

- *prefix-closed* means that for any list satisfying the predicate, all its prefixes must also satisfy the predicate, i.e., $p\ (xs +\!\!+\ ys) \to p\ xs$.
- *overlap-closed* means that for any two overlapped segments satisfying the predicate, their join must also satisfy the predicate, i.e.,

$$(\mathit{length}\ ys > 0 \land p\ (xs +\!\!+\ ys) \land p\ (ys +\!\!+\ zs)) \to p\ (xs +\!\!+\ ys +\!\!+\ zs)$$

This paradigm considers all prefixes of the input list xs in order and calculates the longest suffix satisfying p for each of them. Let $ls(\mathit{pref})$ be the longest suffix of pref satisfying predicate p. For two consecutive prefixes pref_1 and $\mathit{pref}_2 = \mathit{pref}_1 +\!\!+ [v]$, when p is both prefix-closed and overlap-closed, $ls(\mathit{pref}_2)$ must be one of $ls(\mathit{pref}_1) +\!\!+ [v]$, $[v]$ and $[]$.

Algorithm 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 necessary values on the longest valid suffix as *info* (Line 9). When a new element is considered, *lsp* verifies whether $ls(A[0 \dots i-1]) +\!\!+ [A_i], [A_i], []$ are valid, and picks the first valid one among them (Lines 9-18). At this time, combinator *comb* is used to quickly update `info` and verify whether $ls(A[0 \dots i-1]) +\!\!+ [A_i]$ 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.

$$(p \vartriangle \mathit{aux})\ (xs +\!\!+ [v]) = \mathit{comb}\left(v, (p \vartriangle \mathit{aux})\ xs\right)$$

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

Time complexity. Under the efficiency condition, the resulting program runs in $O(n)$ time on a list of length n, as its bottleneck is $O(n)$ invocations of *comb* in the loop.

```
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         merge(info[i], A[rpos[i]], now);
20     }
21     return now.res;
22 }
```

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

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

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

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

- *num* represents the number of segments in the current partition (Line 7).
- $rpos[i]$ represents the index of the right end the ith 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 ith 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 minimum in $xs_L +\!\!+ [v] +\!\!+ xs_R$, i.e., $min\ xs_L > v$ and $min\ xs_R \ge v$.

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

$$\begin{aligned}&\mathit{min}\ xs_L > v \land \mathit{min}\ xs_R \geq v \rightarrow \\ &\qquad (\mathit{orig} \vartriangle \mathit{aux})\ (xs_L +\!\!+ [v] +\!\!+ xs_R) = \mathit{comb}\ (v, ((\mathit{orig} \vartriangle \mathit{aux})\ xs_L, (\mathit{orig} \vartriangle \mathit{aux})\ xs_R))\end{aligned}$$

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

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

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.

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

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.

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

In this paper, we assume that a_0 and $\otimes$ are directly given for two reasons.

- On the one hand, providing a_0 and $\otimes$ is simple for most common update programs. For example, when the update is to add xs_i by an integer, i.e., $u\ (a, w) = a + w$, it is easy to see that a_0 should be 0, and $\otimes$ should be the plus operator, i.e., $a_1 \otimes a_2 = a_1 + a_2$
- On the other hand, finding a_0 and $\otimes$ for a given update program is an isolated synthesis task and thus can be dealt with separately.

Procedure. A segment tree is a tree-like data structure where each vertex corresponds to a segment in the list. On each vertex, several values with respect to the corresponding segment are maintained.

- For each update operation, it 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, it 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 outputs of *h* and the auxiliary program *aux* 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 $\lambda w.u\ (\mathit{tag}[k], w)$, but such an update has not been considered by all nodes in the subtree of node k yet.

There are several functions used in this template:

```
1  struct Info {
2      Int res; // Variable representing the output of h.
3      // Variables representing the output of aux.
4  }info[N];
5  Int tag[N];
6  void apply(int pos, Int a){
7      info[pos] = /*comb2 (a, info[pos])*/;
8      tag[pos] = tag[pos] ⊗ a;
9  }
10 void pushdown(int pos) {
11     apply(pos * 2, tag[pos]);
12     apply(pos * 2 + 1, tag[pos]);
13     tag[pos] = a0;
14 }
15 void initialize(int pos, int *A, int l, int r) {
16     if (l == r) {info[pos] = /*h [A[l]], aux [A[l]]*/; return;}
17     int mid = l + r >> 1;
18     initialize(pos * 2, A, l, mid);
19     initialize(pos * 2 + 1, A, mid + 1, r);
20     info[pos] = /*comb1 (info[pos * 2], info[pos * 2 + 1])*/;
21 }
22 void update(int pos, int l, int r, int ul, int ur, Int a) {
23     if (l > ur || r < ul) return;
24     if (l >= ul && r <= ur) {apply(pos, a); return;}
25     int mid = l + r >> 1; pushdown(pos);
26     update(pos * 2, l, mid, ul, ur, a);
27     update(pos * 2 + 1, mid + 1, r, ul, ur, a);
28     info[pos] = /*comb1 (info[pos * 2], info[pos * 2 + 1])*/;
29 }
30 Info query(int pos, int l, int r, int ul, int ur) {
31     if (l > ur || r < ul) return {/*h [], aux []*/};
32     if (l >= ul && r <= ur) return info[pos];
33     int mid = l + r >> 1; pushdown(pos);
34     return /*comb1 (query(pos * 2, l, mid, ul, ur), query(pos * 2 + 1,
           mid +1, r, ul, ur))*/;
35 }
36 void range(int n, int *A, int m, Operator* op) {
37     initialize(1, A, 0, n - 1);
38     for (int i = 0; i < m; ++i) {
39         if (op[i].type == Update) {
40             update(1, 0, n - 1, op[i].l, op[i].r, op[i].a);
41         } else {
42             print(query(1, 0, n - 1, op[i].l, op[i].r));
43         }
44     }
45 }
```

Fig. 21. The sketch for the algorithmic paradigm of RANGE.

- `apply` deals with an update on all elements in the segment corresponding to node *pos* by updating *info*[*pos*] via combinator $comb_2$ (Line 7).
- `pushdown` applies the tag on node *pos* to its children (Lines 11-12) and then clears it (Line 13).

- `initialize` initializes the information for node *pos* which corresponds to range $[l, r]$. It recurses into two children (Lines 18-19) and merges the sub-results via $comb_1$ (Line 20).
- `update` applies an update $([ul, ur], \lambda w.u\ (a, w))$ to node *pos* which corresponds to range $[l, r]$. If $[l, r]$ does not overlap with $[ul, ur]$, the update will be ignored (Line 23). If $[l, r]$ is contained by $[ul, ur]$, a lazy tag will be put (Line 24). Otherwise, `update` recurses into two children (Lines 26-27) and merges the sub-results via $comb_1$ (Line 28).
- `query` calculates a sub-result for query $[ul, ur]$ by considering elements in node `pos` only. It is implemented similarly to `update`.

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

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

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

Similar to the second paradigm of the longest segment problem (Section 5.2), 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 {*"merge"*, *"update"*} × (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, the resulting program takes linear time to do pre-processing and respond to each operation in logarithmic time.