Divide and Conquer Divide-and-Conquer

Oracle-Guided Inductive Synthesis for Applying D&C-Like Algorithmic Paradigms

RUYI JI, Peking University, China
YINGFEI XIONG[*], 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. However, applying them to optimize existing programs is difficult. Therefore, many research efforts have been devoted to the automatic application of algorithmic paradigms. However, all existing synthesizers of this problem use deductive methods, restrict the syntax of the original programs, and thus involve a great burden on the user. To remove this burden, we study the automatic application of paradigms under an inductive setting where the synthesizer has no access to the syntax of original programs. Such a synthesizer has no requirement on the syntax and thus allows the user to provide original programs in any convenient way.

We notice that the synthesis tasks of applying various paradigms have a similar form as that of D&C. We denote these paradigms as D&C-like paradigms, propose a novel type of synthesis problems, namely *lifting problems*, which unify the tasks of applying D&C-like paradigms, and then propose an efficient inductive solver *AutoLifter* for lifting problems. The main challenge of solving lifting problems is scalability, and we address this issue by divide-and-conquer. *AutoLifter* uses two methods, component elimination and variable elimination, to divide the lifting problem into subtasks and derive simple approximated specifications for these subtasks. In this way, each subtask is tractable by existing inductive synthesis techniques. We analyze this procedure in theory and prove that the usage of the approximated specification does not affect the effectiveness of *AutoLifter*.

We evaluate *AutoLifter* on 93 programming tasks related to 6 different D&C-like algorithmic paradigms. *AutoLifter* solves 82/93 tasks with an average time cost of 6.66 seconds and achieves competitive performance compared with an existing deductive synthesizer for D&C algorithms that requires more input.

1 INTRODUCTION

Efficiency is a major pursuit in practical software design, and designing suitable algorithms is a fundamental way to achieve efficiency. To reduce the difficulty of algorithm design, researchers have proposed many *algorithmic paradigms*, such as divide-and-conquer (D&C), 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 asks us to recursively divide the problem into sub-problems, and combine the solutions to the sub-problems into the solution to the original problem. However, given a concrete problem, how to combine the solutions is totally unknown and up to the developer to discover.

[*]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; 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; 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, kcz@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/

To reduce the burden on the user, many research efforts have been devoted to the automatic application of algorithmic paradigms. These approaches take an (unoptimized) original program as input and apply a specific algorithmic paradigm to optimize the program. 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], and incrementalization [Acar et al. 2005].

However, as far as we are aware, all existing approaches use deductive methods, which restrict the original program and impose a non-trivial burden on the user. In these approaches, the syntax of the original program is available to the system, and the system uses deductive program transformation to solve or simplify the task. To ensure that the program transformation can be applied successfully, these approaches have strict requirements for the original program. For example, systems 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]. The system for incrementalization [Acar et al. 2005] requires that a significant portion of the operations performed in the execution of the original program would not depend on the part of the input that may be changed during an incremental computation, otherwise the synthesized incremental program may not speed up, or even slow down the computation. Satisfying these requirements is typically non-trivial. For example, as will be shown later, applying single-pass requires implementing 39.29%-56.90% of the auxiliary functions needed for applying D&C.

To remove this burden, in this paper we view the problem of applying an algorithmic paradigm as an oracle-guided inductive synthesis (OGIS) problem [Jha and Seshia 2017]. In this problem, the goal is to synthesize, in the target algorithmic paradigm, a program that is equivalent to the original program, and the synthesis is conducted by interacting between a synthesizer and a verifier. The synthesizer has no access to the syntax of the original program but is only able to invoke the original program with some input to obtain the output. The verifier verifies the correctness of the synthesized program and produces counterexamples when it is incorrect. As a result, there is no restriction on how to implement the original program from the synthesizer, and the user is free to choose any (potentially slow) implementation as long as the verifier allows.

The first contribution of this paper is a novel type of synthesis problems that unifies the application of various paradigms, namely lifting problems. Different from previous approaches that aim at a specific paradigm, we observe that the application of various paradigms can be concluded as synthesizing (1) a combination program for combining the solutions of sub-problems and (2) an auxiliary program providing 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 the longest segment problem [Zantema 1992]. We call such algorithmic paradigms as *D&C-like* paradigms, formalize the task of finding the combination program and the auxiliary program as a lifting problem, and provide reductions from the application of the above paradigms to the lifting problem. Through these reductions, a solver for the lifting problem can be instantiated as an inductive synthesizer for applying any of the above paradigms.

The second contribution of this paper is an efficient inductive solver AutoLifter for lifting problems. The core challenge of solving lifting problems is scalability: the programs to be synthesized are too large to be handled by existing synthesizers. To address this challenge, *AutoLifter* decomposes the synthesis target of a lifting problem into a sequence of subprograms, each tractable by an existing synthesizer, and then synthesizes each subprogram one by one. In other words, we divide and conquer the synthesis problem of applying D&C-like paradigms.

However, the subprograms closely depend on each other, and it is difficult to derive a precise specification for an individual subprogram for independent synthesis. We address this issue by approximation. We propose two decomposition methods that derive simple but possibly imprecise specifications for each subprogram. In this way, we enable efficient synthesis for each subprogram. Furthermore, we ensure that the loss introduced by the approximated specification is ignorable.

- We guarantee the soundness of *AutoLifter* by always using the precise specification for the last subprogram. If any previous step produces an incorrect subprogram under the weak specification, the last step would fail and would not lead to an incorrect result.
- We further prove that on a random lifting problem, the probability for this synthesis failure is small enough to be ignored.

The third contribution of this paper is an evaluation of the performance of AutoLifter. We instantiate *AutoLifter* as six 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 mentioned before. We construct a dataset of 93 tasks for applying these paradigms, which are collected from existing datasets [Farzan and Nicolet 2017, 2021b], 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 93 tasks with an average time cost of 6.66 seconds, significantly outperforming existing inductive synthesizers that can be applied to lifting problems.
- For D&C algorithms, *AutoLifter* achieves competitive, or even better, performance compared with an existing deductive approach that requires more input from users.

2 MOTIVATING EXAMPLES

In this section, we illustrate the limitation of existing approaches and the basic ideas of *AutoLifter*.

2.1 Divide-and-Conquer for Second Minimum

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

Fig. 1. Second minimum

Let *sndmin* be a function returning the second-smallest value in an integer list. Figure 1 shows a natural implementation of this function (in Python-like syntax). It sorts the input list *xs* ascendingly via `sorted` and returns the second element (0-indexed) via `[1]`. Furthermore, it returns a large enough number `INF` when the length of *xs* is smaller than 2. This program is not efficient, taking $O(n \log n)$ time.

Manually applying D&C. Suppose we would like to get an efficient program for *sndmin* by applying D&C, a paradigm widely used for optimization. In D&C, we need to decompose the 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 $sndmin\ xs$. Therefore, we need to find a combinator *comb* such that the following holds, 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 minimums of the sub-lists are not enough to determine the second minimum of the whole list. To solve this problem, a standard way is to find an auxiliary program *aux* to extend the original program *sndmin* into $sndmin'\ xs = (sndmin\ xs, aux\ xs)$, such that a combinator exists for $sndmin'$, i.e.,

$$sndmin'\,(xs_L +\!\!+\, xs_R) = comb\,(sndmin'\ xs_L, sndmin'\ xs_R), \textbf{where } sndmin'\,xs = (sndmin\,xs, aux\,xs) \quad (1)$$

```
def dac(xs, l, r):
  if l + 1 <= r:
    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. 2. A divide-and-conquer template on lists

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

Fig. 3. A single-pass program for *sndmin*.

For this problem, we can use an auxiliary program that returns the first minimum and a combinator that produces the second and the first minimums from those of the sub-lists, as follows.

$$aux\ xs = minimum\ xs \qquad \begin{aligned} &comb\ \left((sndmin_L, aux_L), (sndmin_R, aux_R)\right) \\ &\quad = (\min(sndmin_L, sndmin_R, \max(aux_L, aux_R)), \min(aux_L, aux_R)) \end{aligned}$$

By filling the above two programs into the D&C template in Figure 2, we obtain a D&C algorithm for *sndmin*. Here *orig* stands for the original program *sndmin*. Note that though in this template *aux* is only applied to the list of length 1, we define it for any list to guide the design of the *comb*. This program takes $O(n)$ time[1], and with proper parallelization, the complexity can be reduced to $O(n/p)$, where $p \leq n/\log n$ is the number of processors.

As we can see from this process, applying D&C is non-trivial. Though the template in Figure 2 is standard for D&C algorithms on lists, we still need to figure out how to write the auxiliary program *aux* and the combinator *comb*. The implementations of the two programs are observably much more complex than the original program in Figure 1.

An existing deductive approach. *ParSynt* [Farzan and Nicolet 2017, 2021b] is an existing approach that automatically applies D&C to the original program. As mentioned in the introduction, *ParSynt* requires the original program to be written as *single-pass*, which enumerates elements in the input list only once. Figure 3 shows a single-pass implementation for *sndmin*.

As we can see, the single-pass implementation has already included an auxiliary variable to record the first minimum, and calculates the first and second minimums in each iteration. In other words, the user has finished most of the difficult work in applying D&C, and the help provided by *ParSynt* is limited. As we shall show in Section 8, the auxiliary values required for a single-pass implementation account for 39.29%-56.90% of those required by D&C. Besides, implementing programs in single-pass is also error-prone: the dataset used by Farzan and Nicolet [2021b] contains two bugs introduced when the author manually implements the original programs into single-pass[2].

2.2 Incrementalization for Second Minimum

In the second example, we consider the application of another paradigm *incrementalization*.

Manually applying incrementalization. Suppose now we have computed the second minimum for a list *xs*, then the list is changed, and we would like to quickly determine the second minimum for the new list. For simplicity, let us assume the change is to append an element v to the list. Incrementalization tells us that we should compute some auxiliary value from the previous list, such that the second minimum and the auxiliary value of the changed list can be quickly obtained. In other words, we need to find a program *aux* to compute the auxiliary value and a program *comb*

[1]In this paper, we assume *orig* and *aux* take $O(1)$ time on a list of length one.
[2]Confirmed by the authors

to quickly update the result and the auxiliary value.

$$\mathit{sndmin}'\ (\mathit{append}\ \mathit{xs}\ v) = \mathit{comb}\ (\mathit{sndmin}'\ \mathit{xs}, v), \textbf{where } \mathit{sndmin}'\ \mathit{xs} = (\mathit{sndmin}\ \mathit{xs}, \mathit{aux}\ \mathit{xs}) \qquad (2)$$

We can derive the following programs, where each update from *comb* takes only $O(1)$ time.

$$\mathit{aux}\ \mathit{xs} = \mathit{minimum}\ \mathit{xs} \qquad \begin{aligned} &\mathit{comb}\ ((\mathit{sndmin}_{pre}, \mathit{aux}_{pre}), v) \\ &\quad = (\min(\mathit{sndmin}_{pre}, \max(\mathit{aux}_{pre}, v)), \min(\mathit{aux}_{pre}, v)) \end{aligned}$$

Note that this task is similar to implementing *sndmin* as single-pass, as both of them are concerned with the second minimum after a new element is appended. Similar to the case of single-pass, applying incrementalization is also challenging, as we need to define a proper auxiliary function and update the result and the auxiliary value at each change.

An existing deductive approach. Acar et al. [2005] proposed an approach for automatic incrementalization. In the algorithm generated by this approach, the auxiliary value is a trace of the evaluated operations during the execution of the original program, and the combinator selects and re-evaluates the operations that are affected by the changed part of the input. As a result, the effectiveness of this approach depends on how the original program is implemented. For example, if the natural implementation of *sndmin* (Figure 1) is provided, the generated algorithm will trace into the sorting function `sorted` and will result in a time complexity of $O(\log n)$ to deal with each change, significantly worse than the $O(1)$ time of the ideal algorithm. Using the single-pass program in Figure 3 as the original program could lead to an $O(1)$ incremental algorithm, but the original program already contains an incremental algorithm of the same complexity.

2.3 An Overview of *AutoLifter*

The interface. We first demonstrate how an end user uses our approach for the two previous examples. As mentioned in the introduction, *AutoLifter* can be instantiated as different synthesizers for different algorithmic paradigms. Currently, we have implemented a set of synthesizers based on *AutoLifter*, including two synthesizers for applying D&C and incrementalization on lists. The user only needs to pick the respective synthesizer and feed the synthesizer with the original program to be optimized. There is no restriction on the provided programs, the user could use any natural implementation of *sndmin*, such as the one in Figure 1. When this program is used, our synthesizers take 0.21 seconds and 0.18 seconds to produce the ideal algorithms, which take $O(n/p)$ time in parallel and $O(1)$ time per update, respectively.

The lifting problem. Next, we discuss in more detail how our approach works. We can see that the previous two examples share some commonalities. First, in both examples, the core of the problem is to find two programs, *aux* and *comb*. Second, the equations the two programs have to satisfy, Formula 1 and 2, have a similar structure: they both use *aux* to produce some auxiliary values besides the original result such that the operation performed by *comb* is possible.

Based on the above observations, we uniformly define the core problems of the two examples as a new type of synthesis problems for synthesizing the two programs *aux* and *comb*. We call such a problem a *lifting* problem since the *aux* function lifts the original program to make *comb* possible.

A lifting problem is defined as a syntax-guided synthesis (SyGuS) problem [Alur et al. 2013], and the grammars for *comb* and *aux* are useful in controlling the complexity of the generated algorithm. For example, if we include only constant-time programs in the grammar for *comb*, we can ensure the synthesized *comb* to be constant-time, and thus the D&C algorithm takes $O(n/p)$ time, and the incremental computation can be performed in constant time.

The scalability challenge. The main challenge of the lifting problem is scalability. Let us consider another example, *maximum segment sum (mss)* [Bird 1989b]. Given an integer list *xs*, the task of

```
        def aux(xs):
a@mps     mps = max([sum(xs[:i+1])
                     for i in range(lens(xs))])
a@mts     mts = max([sum(xs[i:])
                     for i in range(lens(xs))])
a@sum     sumx = sum(xs)
          return (mps, mts, sumx)
        def comb(L, R):
          (mssL, (mpsL, mtsL, sumL)) = L
          (mssR, (mpsr, mtsR, sumR)) = R
c@mss     mss = max(mssL, mssR, mtsL + mpsR)
c@mps     mps = max(mpsL, sumL + mpsR)
c@mts     mts = max(mtsL + sumR, mtsR)
c@sum     sumx = sumL + sumR
          return (mss, (mps, mts, sumx))
```

Fig. 5. D&C for maximum segment sum.

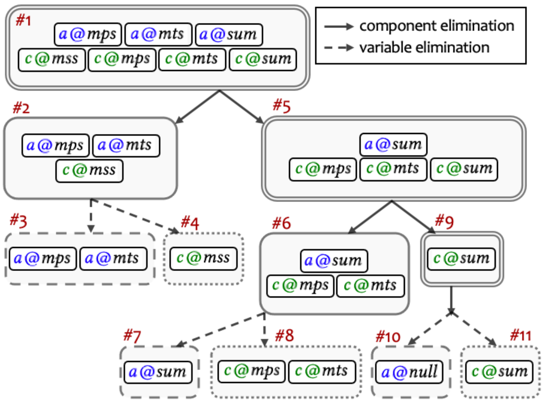

Fig. 6. The synthesis procedure of *AutoLifter*.

mss is to select a non-empty segment (i.e., consecutive subsequence) *s* from *xs* such that the sum of its elements is the largest among all segments, and return this sum.

Figure 4 shows a natural implementation of *mss*. This implementation enumerates all possible segments, calculates the sums of their elements, and selects the maximum sum, taking $O(n^3)$.

```
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. 4. Maximum segment sum

Suppose now we would like to optimize this program into $O(n/p)$ using D&C. Figure 5 shows one possible implementation of $(aux, comb)$ in a desirable D&C algorithm, where some statements are tagged for easy reference. This algorithm uses three auxiliary values. The first two, the maximum prefix sum (*mps*) and the maximum tail sum (*mts*) are used to enable the calculation of the original result *mss*. The last one, *sumx*, is used to enable the calculation of the first two auxiliary values.

The scale of these two programs is beyond the scalability of existing inductive synthesizers. In the grammars we used, these two programs take at least 28 operators to implement. In comparison, a state-of-the-art synthesizer for list-related programs, *DeepCoder* [Balog et al. 2017], times out on 40% tasks in a dataset for list-related programs with 5 operators with a time limit of one hour.

We address the scalability issue via a decomposition system that divides the task of synthesizing the whole program into smaller subtasks, each synthesizing a subprogram. To get an efficient synthesizer, the key point in this system is to get independent specifications for each subtask. However, getting independent specifications is difficult in the lifting problem, where subprograms closely depend on each other. For example, let us consider *c@mss* and *a@mps* in Figure 5.

- On the one hand, *c@mss* cannot be synthesized before *a@mps* because the input of *c@mss* is defined by *a@mps*.
- On the other hand, *a@mps* cannot be synthesized before *c@mss* because the output of *a@mps* is consumed by *c@mss*.

As a result, though the programs in Figure 5 are formed by many simple components, we cannot directly decompose them due to the dense dependency in the lifting problem.

The main idea. To get independent specifications, our main idea is to approximate the precise specification. In general, a decomposition aims at a specification $\varphi(f, g)$ where the synthesis target is formed by two dependent subprograms f and g. Its goal is to synthesize these two subprograms separately. Performing decomposition requires an independent specification for at least one program and thus is challenging when f and g are dependent. Note that though specification $\exists g \varphi(f, g)$ is always a precise and independent specification for f, it is not feasible because it contains a second-order existential quantification $\exists g$ and is not supported by existing synthesizers.

To address this issue, our main idea is to use an imprecise specification to specification $\tilde{\varphi}(f)$ to approximate $\exists g \varphi(f, g)$. At this time, we can conduct the decomposition as below.

(1) Synthesize the first subprogram f from the approximated specification $\tilde{\varphi}(f)$.
(2) Synthesize a corresponding g from $\varphi(f, g)$ by substituting the synthesized f.

In this way, the original task is decomposed into two simpler tasks, when $\tilde{\varphi}$ is simple enough.

The synthesis process. Figure 6 shows the procedure for *AutoLifter* to synthesize the programs in Figure 5[3], where the nodes are subtasks, the arrows indicate task decomposition, the tags within the nodes indicate the subprograms synthesized by the subtasks, and different line styles indicate different decomposition/subtask types. After the decomposition, the scale of the leaf-subtasks (#3, #4, #7, #8, #10, #11) is greatly reduced compared with the original task (#1), and thus these subtasks can be solved by existing inductive synthesizers. There are two decomposition methods.

- *Component elimination.* As we have seen, the *comb* and *aux* programs can be divided into two groups of components, where c@*mss*, a@*mps* and a@*mts* are used for calculating the result of the original program, while the others are used for calculating the introduced auxiliary values. Therefore, we decompose the original task (#1) into two subtasks, one for calculating the original result (#2) and one for calculating the introduced auxiliary values (#5), i.e., some components needed for #1 are eliminated in #2. Similarly, #5 can be further decomposed as one for calculating the auxiliary values introduced in the previous step (#6) and one for calculating newly introduced auxiliary values (#9). #9 does not need to be further decomposed as the synthesized auxiliary program is *null*, indicating no auxiliary value is needed.
- *Variable elimination.* In each subtask decomposed from the previous method, we still need to synthesize the components both for *comb* and *aux*. We further decompose each subtask into two subtasks, one for finding auxiliary values that ensure the existence of a combination function (e.g., #3), and one for finding a program that implements the function (e.g., #4).

In each decomposition, the specification used for the first task is not precise as it does not ensure the second task is solvable. For example, in task #2, we are concerned only about how to calculate *mss* but do not require that the synthesized auxiliary values can be calculated (task #5). Similarly, in task #3, we are concerned with only the existence of a combinator function, but we do not require that the function can be implemented in the respective grammar (task #4).

Properties. Though the use of an approximated specification may result in an incorrect subprogram being synthesized from the first subtask, it harms only the completeness but not the soundness of the whole system. In the second subtask of each decomposition, we always use a precise specification involving also the synthesis result of the first subtask. So, if an incorrect subprogram is synthesized in the first subtask, the second subtask would fail and would not produce an incorrect result.

Moreover, to ensure the practical performance of *AutoLifter*, we perform a theoretical analysis of the probability for *AutoLifter* to fail due to the approximated specification. We prove that this probability is bounded when (1) the synthesizer for the first subtask is an Occam solver [Ji et al. 2021], and (2) the approximated specification is weaker than the precise one and meanwhile provides a strong enough constraint that any two programs can hardly satisfy it at the same time. We design *AutoLifter* under the guidance of this theoretical result and thus ensure that *AutoLifter* seldom fails due to the approximated specification, which is justified by our evaluation.

[3]In *AutoLifter*, programs are written in domain-specific languages instead of the syntax in Figure 5. For example, a@*mps* is implemented as *maximum (scanr (+) xs)*. Details on the languages we used can be found in Section 7.

3 LIFTING PROBLEMS

3.1 Notations

In this paper, we regard a type as a set and use the two terms interchangeably. We distinguish two kinds of types: *bounded types* where the values are of constant size (e.g., `Int`, `Bool`, and their tuples), and *unbounded types* where the size of values can be arbitrarily large (e.g., `List`, `Tree`, and tuples including them). For simplicity, we assume any function whose domain is a bounded type is implemented as a constant-time program. Though this assumption does not hold in general (e.g., calculating the largest prime number smaller than n), it is not an issue in synthesizing algorithms over data structures as we usually only apply constant-time operators on bounded values.

We use uppercase letters such as A, B to denote types, lowercase letters such as a, f to denote values and functions, and overline letters such as $\overline{a}$ to represent tuples. We use T^n to represent n-arity product $T \times \cdots \times T$ of type T, $f_1 \vartriangle f_2$ to apply two functions to the two components in a pair respectively, $f_1 \times f_2$ to apply two functions to the same value, and f^n to apply function f to each component in an n-tuple, as shown below.

$$(f_1 \vartriangle f_2)\ x = (f_1\ x, f_2\ x) \qquad (f_1 \times f_2)\ (x_1, x_2) = (f_1\ x_1, f_2\ x_2) \qquad f^n\ (x_1, \dots, x_n) = (f\ x_1, \dots, f\ x_n)$$

Besides, to distinguish between synthesis targets, quantified values, and known functions/values in a specification, we mark them as red, green, and black respectively in the rest of this paper.

3.2 Lifting Problem

In Section 2, we have seen two synthesis tasks (Formula 1 and 2). Each task is under a scenario that (1) a list is created via an operator from some existing lists ($xs_L +\!\!+\ xs_R$ and $append\ xs\ v$), and (2) the output of an original program ($sndmin$) on the created list is calculated. Both tasks aim at finding an auxiliary program and a combinator such that the output of concern can be computed from the outputs on existing lists. We conclude synthesis tasks in this form as the lifting problem.

Definition 3.1 (Lifting Problem). Given an original program $orig$, an operator op, and two languages $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$, lifting problem $\mathsf{LP}(orig, op, \mathcal{L}_{aux}, \mathcal{L}_{comb})$ is to find an auxiliary program $aux \in \mathcal{L}_{aux}$ and a combinator $comb \in \mathcal{L}_{comb}$ to make the diagram on the right commute, i.e., all paths of the same start and the same end represent the same function, where id is the identity function $id\ x = x$.

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

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

In the diagram, A is the domain of $orig$, B_{orig} is the range of $orig$, arity n is the number of A-typed inputs taken by the operator op, and complementary type C supplies the remaining inputs to op. The table below associates these notations with the two tasks discussed in Section 2, where `Unit` is a singleton type, and () is the only element in `Unit` that provides no information.

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

In the sense of algorithm optimization, the lifting problem seeks a method to eliminate the construction of the A-element. The upper-left path in the diagram constructs an intermediate A-element via op and then immediately consumes it via $orig$. In comparison, the lower-right path in the diagram avoids this construction by directly calculating from existing results, via the synthesized combinator $comb$. This intuition matches a general optimization strategy, *fusion* [Pettorossi and

Proietti 1996], which suggests that a program is efficient if there is no unnecessary intermediate data structure that is produced and consumed during the computation.

In this paper, we assume that DSL $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ are implicitly given and thus abbreviate a lifting problem as $\mathsf{LP}(orig, op)$. Besides, we assume that $\mathcal{L}_{aux}$ contains a constant function *null* that maps anything to the unit constant $()$. It represents the case where no auxiliary value is required.

Requirements. The performance of our approach depends on two additional requirements.

(1) To synthesize a program with a target time complexity, the DSL $\mathcal{L}_{aux}$ and $\mathcal{L}_{comb}$ should contain only sufficiently efficient programs to establish this time complexity in the corresponding template. For example, to achieve $O(n/p)$ complexity with the D&C template in Figure 2, $\mathcal{L}_{comb}$ should contain only constant-time programs.
(2) To bound the failure rate of *AutoLifter* caused by the approximated specifications, the original program *orig* and the programs in $\mathcal{L}_{aux}$ need to be *compressing*, i.e., there are sufficiently many inputs that have the same output. Details on the relation between compressing and the failure rate can be found in Section 5.2. Algorithms on data structures are usually compressing because they usually summarize a result from a possibly infinite input data structure.

In this paper, we establish the two requirements by focusing our theoretical analysis and implementation on a subclass of the lifting problem where the complementary type C, the output type of the original program *orig*, and the output type of each $aux \in \mathcal{L}_{aux}$ are bounded types. This subclass already includes many interesting algorithms such as all examples we have seen before. We can see that this subclass ensures both requirements.

(1) In this subclass, the input type of *comb* is bounded, and thus *comb* must be constant-time as we assume all programs on bounded values are constant-time. In this way, we ensure the efficiency of the synthesized algorithm for all paradigms considered in this paper, e.g., the synthesized D&C algorithm takes $O(n/p)$ time in parallel. Details on the efficient guarantees can be found in Section 8.1.
(2) In this subclass, both *orig* and *aux* map an unbounded type to a bounded type, and thus there are sufficiently (actually infinitely) many inputs whose outputs are the same.

Though we only conduct the theoretical analysis on the above subclass for simplicity, we believe our reasoning process is general and our approach can be applied to other cases where the two requirements are satisfied. Conducting theoretical analysis on these cases is future work.

3.3 More Applications of the Lifting Problem

We have seen how the applications of D&C and the specialized version of incrementalization are reduced to the lifting problem. It is also not difficult to see that applying the specialized version of incrementalization is equivalent to rewriting the program into single-pass, which is the required input format for *ParSynt* [Farzan and Nicolet 2017, 2021b]. In this subsection, we demonstrate two more applications, the general version of incrementalization and the segment trees.

Generalized Incrementalization. In the previous version of incrementalization, the only allowed change is to append an element to the end of the list. In a general version, there can be different types of changes. To capture that, we introduce an operator $change : C \times A \mapsto A$ which updates an A-element with some change operation in type C. Incrementalization seeks a combinator *comb* calculating the output of *orig* from the output before the change. This task can be regarded as a lifting problem $\mathsf{LP}(orig, op)$ where $op\ (c, (x)) = change\ (c, x)$.

Segment Trees. This application demonstrates the case where multiple combinators are needed. A segment tree is a data structure for efficiently answering queries about a specific property of a segment in a long list. Given a list, after a linear-time pre-processing, a segment tree can evaluate a

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

(a) Generalized lifting problem

$$\begin{tikzcd}
A \arrow[rr, "orig"'] & & B_{orig} \\
C \times A^n \arrow[u, "op"] \arrow[rr, "id \times (aval \triangle aux_1)^n"] & & C \times B_1'^n \arrow[u, "comb_1"']
\end{tikzcd}$$

(b) Subtask 1

$$\begin{tikzcd}
A \arrow[rr, "aux_1 \triangle aux_2"'] & & B_{aux} \\
C \times A^n \arrow[u, "op"] \arrow[rr, "id \times ((aval \triangle aux_1) \triangle aux_2)^n"] & & C \times B_2'^n \arrow[u, "comb_2"']
\end{tikzcd}$$

(c) Subtask 2 (simplified)

Fig. 7. The commutative diagrams describing the input task and the two subtasks of component elimination.

pre-defined function *orig* on a segment (e.g., "answer the second minimum of the segment from the 2nd to the 500th 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 500th element by 1"), each in $O(\log n)$ time.

In brief, a segment tree (1) uses D&C to respond to queries, and (2) uses incrementalization to respond to changes. Therefore, implementing a segment tree for a specific query is to find three programs, $comb_1$, $comb_2$ and *aux* such that $(comb_1, aux)$ is a solution to the lifting problem of D&C and $(comb_2, aux)$ is a solution to the lifting problem of incrementalization.

Though two combinators and two constraints are involved in this task, they can be unified into a lifting problem $\mathsf{LP}(orig, op)$ by introducing a boolean variable *tag* to the operator, as shown below.

$$op\ ((tag, c), (xs, ys)) = \textbf{if}\ tag\ \textbf{then}\ xs +\!\!+\ ys\ \textbf{else}\ change\ (c, xs)$$

This constructed operator is equivalent to those of D&C and incrementalization when *tag* is fixed to *true* and *false*, respectively.

Details on segment trees and another application of the lifting problem to the longest segment problem [Zantema 1992] can be found in Appendix C.

4 *AUTOLIFTER*

AutoLifter decomposes the lifting problem into subtasks via two methods, *component elimination* and *variable elimination*. Both methods use an approximated specification to enable the decomposition.

4.1 Decomposition Method 1: Component Elimination

As mentioned in Section 2.3, component elimination decomposes the target program $(aux, comb)$ into two subprograms $(aux_1, comb_1)$ and $(aux_2, comb_2)$, where $(aux_1, comb_1)$ is used to calculate the original result, $(aux_2, comb_2)$ is used to calculate the introduced auxiliary values, and $(aux, comb) = (aux_1 \triangle aux_2, comb_1 \triangle comb_2)$. In other words, some components of *aux* and *comb* are eliminated in the subtasks. In the next iteration, component elimination is recursively applied to decompose $(aux_2, comb_2)$. To unify the recursive applications, we introduce a generalized version of the lifting problem, as shown in Figure 7a, where the usage of *orig* on the bottom is replaced with a separate program *aval*. We can obtain the original lifting problem by setting *aval* to *orig*.

To synthesize $(aux_1, comb_1)$, the approximated specification is shown in Figure 7b. It requires calculating only the original result and is obtained by keeping only the *orig* component on the top line in the specification for the lifting problem. Note that this approximated specification is weaker than the precise specification of $(aux_1, comb_1)$ because we do not require aux_1 to be calculable, i.e., do not require the existence of $(aux_2, comb_2)$ in the respective DSLs.

After we get $(aux_1, comb_1)$ by synthesizing from the approximated specification, we can substitute them to get the specification for $(aux_2, comb_2)$. Since we already know that the specification in Figure 7b is satisfied, we can simplify the substituted specification by removing $comb_1$ and the *orig* component on the top, as shown in Figure 7c[4]. We can see this specification is in the form of the generalized lifting problem in Figure 7a, where $(orig, op, aval)$ is set to $(aux_1, op, aval \vartriangle aux_1)$, and thus can be further decomposed by component elimination.

Example 4.1. In the synthesis procedure shown in Figure 6, there are three generalized lifting problems (task #1, #5, #9), and component elimination is applied to each of them. We list the parameters of these tasks in the right-side table, and their results have been shown in Figure 6.

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

4.2 Decomposition Method 2: Variable Elimination

Variable elimination aims at decomposing the first subtask generated by component elimination (Figure 7b). It naturally decomposes $(aux_1, comb_1)$ into aux_1 and $comb_1$, and synthesizes aux_1 first.

The approximated specification for aux_1 requires the existence of a combinator function. A function is a binary relation where each input corresponds to only one output so that the outputs of *comb* on two equal inputs must also be equal. Therefore, we get a specification for aux_1, that is, aux_1 needs to satisfy the following formula for any $c \in C$ and any $\overline{a}, \overline{a'} \in A^n$.

$$(aval \vartriangle aux_1)^n\ \overline{a} = (aval \vartriangle aux_1)^n\ \overline{a'} \to orig\ (op\ (c, \overline{a})) = orig\ (op\ (c, \overline{a})) \tag{3}$$

This specification is weaker than the precise specification of aux_1 because it only concerns the existence of the *comb* function but does not require that the function can be implemented in $\mathcal{L}_{comb}$.

Variable elimination uses this formula in the first subtask, synthesizes aux_1 from it, and then substitutes aux_1 in the original specification for synthesizing $comb_1$.

Example 4.2. In Figure 6, task #2 is decomposed into task #3 and #4 by variable elimination. The formula below shows the weaker specification in this application, where $mss' = mss \vartriangle aux_1$.

$$(mss'\ xs_L, mss'\ xs_R) = (mss'\ xs'_L, mss'\ xs'_R) \to mss\ (xs_L +\!\!+ xs_R) = mss\ (xs'_L +\!\!+ xs'_R)$$

This specification is effective in excluding incorrect auxiliary programs. For example, when aux_1 returns only the maximum prefix sum, this specification is violated when xs_L, xs_R, xs'_L, xs'_R are set to lists $[1], [1], [1, -1], [1]$ respectively. At this time, the specification is evaluated as below.

$$((1,1),(1,1)) = ((1,1),(1,1)) \to 2 = 1$$

4.3 Workflow of *AutoLifter*

So far, we have introduced the two decomposition methods individually. Algorithm 1 shows how *AutoLifter* uses these two methods to decompose the lifting problem.

(1) Given a lifting problem, *AutoLifter* first regards it as a generalized lifting problem (Line 11) and then decomposes it via component elimination (Lines 6-10). The first subtask is solved by variable elimination (Line 7), and the second subtask is solved recursively (Line 9). The recursion terminates when no new auxiliary value is required (Line 8).

(2) *AutoLifter* applies variable elimination to further decompose the first subtask generated by component elimination (Lines 1-5) and solves the two subtasks via client synthesizers (Lines 3, 5). We will discuss the implementations of these client synthesizers in Section 6.

Example 4.3. Figure 6 shows one possible procedure for Algorithm 1 to solve the lifting problem corresponding to D&C and maximum segment sum.

[4]In this procedure, we adjust the input structure of $comb_1$ and $comb_2$ to get a simpler specification. It does not affect the synthesis task, and thus we omit it for simplicity.

Algorithm 1: The synthesis framework of *AutoLifter*.

```
Input: A lifting problem LP(orig, op) and two client synthesizers $\mathcal{S}_{aux}$ and $\mathcal{S}_{comb}$.
Output: A solution (aux, comb) to lifting problem LP(orig, op).
1  Function VariableElimination(orig, op, aval):
2      $\tilde{\varphi} \leftarrow$ the approximated specification generated by variable elimination (Formula 3);
3      if null is valid for $\tilde{\varphi}$ then $aux_1 \leftarrow null$ else $aux_1 \leftarrow \mathcal{S}_{aux}(\tilde{\varphi})$;
4      $\varphi' \leftarrow$ the specification for a combinator corresponding to $aux_1$(Figure 7b);
5      return $(aux_1, \mathcal{S}_{comb}(\varphi'))$;
6  Function ComponentElimination(orig, op, aval):
7      $(aux_1, comb_1) \leftarrow$ VariableElimination(orig, op, aval);
8      if $aux_1 = null$ then return $(null, comb_1 \vartriangle null)$;
9      $(aux_2, comb_2) \leftarrow$ ComponentElimination($aux_1$, op, aval $\vartriangle$ $aux_1$);
10     return (aux, comb) constructed from $(aux_1, comb_1)$ and $(aux_2, comb_2)$;
11 return ComponentElimination (orig, op, orig);
```

5 PROPERTIES OF *AUTOLIFTER*

In this section, we discuss the properties of *AutoLifter*. Due to the space limit, we introduce only the main idea of our analysis and leave the formalizations to Appendix A.

5.1 Soundness and Incompleteness

- *AutoLifter* is sound. Its result must satisfy the specification when the client synthesizers are sound. This is because we use the original specification in the second subtask of each decomposition.
- *AutoLifter* is not complete and may fail on a solvable task. The first subprogram synthesized from the approximated specification may not satisfy its precise specification. At this time, there exist no corresponding second subprogram, and thus our synthesis will fail.

5.2 Probabilistic Completeness

In this section, we shall demonstrate that, in *AutoLifter*, the probability of synthesis failures caused by approximated specifications is small enough to be ignored. Because the incompleteness of *AutoLifter* comes from each decomposition, we first analyze the general form of decomposition, then analyze the decomposition methods we used, and at last, get the conclusion for *AutoLifter*.

General form of decomposition. Continuing to the notations introduced in Section 2.3, we use $\varphi(f, g)$ to denote a specification where the synthesis target is formed by two subprograms f and g, $\tilde{\varphi}(f)$ to denote the approximated specification of the first subprogram.

For simplicity, we further assume the approximated specification is a necessary condition of the precise one, i.e., $\forall f (\exists g \varphi(f, g) \Rightarrow \tilde{\varphi}(f))$. We denote such an approximation as a *weak specification*. As discussed in Section 4.1 and 4.2, the approximated specifications used in our decomposition methods are both weak specifications.

As mentioned in Section 5.1, our decomposition method will fail when the result f synthesized from the weak specification does not satisfy its precise specification. To measure the probability of this failure, we assume that the specification φ is drawn from a distribution, and then the probability can be naturally defined as the following.

$$\Pr[\forall g \neg\varphi(f^*, g) \mid \varphi \text{ is solvable}], \textbf{where } f^* \text{ is the program synthesized for } f$$

In general, directly evaluating the above probability is difficult. Its definition involves second-order quantifier $\forall g$ and the concrete result f^* of the synthesizer. Both of them bring great challenges

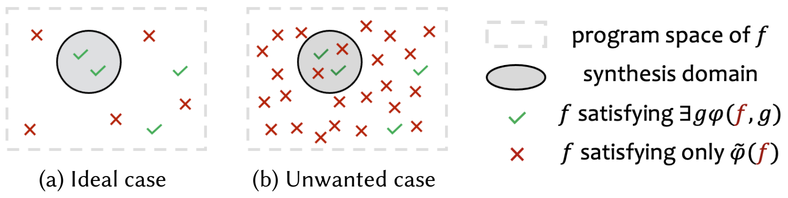
program space of f
synthesis domain
f satisfying $\exists g \varphi(f, g)$
f satisfying only $\tilde{\varphi}(f)$
(a) Ideal case
(b) Unwanted case

Fig. 8. Two possible cases while synthesizing from the weak specification via an Occam solver. The precise specification of f must be satisfied in the first case and may be violated in the second case.

to analysis. To ease the evaluation, we derive a sufficient condition that bounds the probability of incompleteness without involving these two components.

In our sufficient condition, we borrow the concept of *Occam solvers* from a previous study on the generalizability of programming-by-example [Ji et al. 2021]. In brief, for any specification, let s be the size of the smallest valid program, then an α-Occam solver with constant k ensures that the size of the synthesized program is no larger than ks^{α} whatever the concrete specification is.

We consider a sufficient condition for our decomposition method to synthesize successfully when an Occam solver is used, that is, there is no program that (1) has a size no larger than ks^{α}, and (2) satisfies only the weak specification. We visualize this condition in Figure 8a, where the gray circle represents those programs smaller than ks^{α}, i.e., those programs possibly returned by the Occam solver, ticks represent programs satisfying the precise specification, and crosses represent the programs satisfying only the weak specification. Intuitively, this condition requires that there is no cross in the gray circle, and thus the synthesized f in the first step must correspond to a tick and thus must lead to a successful synthesis.

To ensure probabilistic completeness, it is sufficient to ensure this condition is seldom violated. Here, a related factor is the strength of the weak specification, i.e., the probability for a program to satisfy the weak specification. As shown in Figure 8b, when the weak specification is too weak, there will be too many crosses, and thus it is probable that some crosses will be inside the circle.

We prove that ensuring the strength of the weak specification is already sufficient to limit the probability of incompleteness when an Occam solver is used. As shown in the following theorem, our decomposition method seldom fails when (1) an *Occam solver* is used to synthesize the first subprogram f from the weak specification, and (2) the weak specification is strong enough that any pair of programs can hardly satisfy the weak specification at the same time.

THEOREM 5.1 (OCCAM-DECOMPOSITION). *There exists a constant c such that, for any constant $\beta \geq 0$, the probability for our decomposition method to be incomplete, i.e., fail on a random solvable specification, is at most $c \cdot \beta$ when the following two conditions hold.*

- *The synthesizer used for synthesizing f from the weak specification is a complete Occam solver.*
- *For any two different programs f and f^*, the probability for f to satisfy the weak specification $\tilde{\varphi}$ is small enough when f^* satisfies $\tilde{\varphi}$, i.e., probability $\Pr[\tilde{\varphi}(f) \mid \tilde{\varphi}(f^*)]$ is at most β.*

The condition provided by the Occam-decomposition theorem contains neither the second-order quantifier nor the concrete behavior of the synthesizer and thus is easier to reason with.

Decomposition Rules in *AutoLifter*. The sufficient condition in *Occam-decomposition theorem* is formed by two parts. Now, we show that the weak specifications we used in component elimination and variable elimination are strong enough to satisfy the second part.

Our discussion is based on a model of the distribution of specifications. We build up this model by assuming the semantics of each related program (including *orig*, *aval*, *op*, and those programs in $\mathcal{L}_{\text{aux}}$ and $\mathcal{L}_{\text{comb}}$) to be independently random (mapping any input to an independently random output) and perform theoretical analysis under this random assumption. Recall that in Section 3, we assume the input domain of *orig* is unbounded to ensure the compressing property. Here, to ease the analysis, we simplify this assumption by assuming that (1) there is a size limit to bound the input domain to be finite, and (2) the size limit can be arbitrarily large.

As shown in the following two lemmas, both weak specifications used in the two decomposition methods are strong enough for satisfying the condition of the Occam-decomposition theorem.

LEMMA 5.2 (COMPONENT ELIMINATION). *Let $\tilde{\varphi}(aux_1, comb_1)$ be the weak specification in component elimination (Figure 7b). For any $\epsilon > 0$, there exists a threshold, such that when the size limit on the input domain is larger than the threshold, $\Pr[\tilde{\varphi}(aux_1, comb_1) \mid \tilde{\varphi}(aux_1^*, comb_1^*)] \leq \epsilon$ for any two different pairs of programs $(aux_1, comb_1)$ and $(aux_1^*, comb_1^*)$.*

LEMMA 5.3 (VARIABLE ELIMINATION). *Let $\tilde{\varphi}(aux_1)$ be the weak specification in variable elimination (Formula 3). For any $\epsilon > 0$, there exists a threshold such that, when the size limit on the input domain is larger than the threshold, $\Pr[\tilde{\varphi}(aux_1) \mid \tilde{\varphi}(aux_1^*)] \leq \epsilon$ for any two different programs aux_1, aux_1^*.*

The compressing property we assumed in Section 3 is crucial to the effect of variable elimination. It requires the original program *orig* and the programs in $\mathcal{L}_{aux}$ to be compressing, i.e., there are sufficiently many inputs that have the same output. On the one hand, the proof of Lemma 5.3 relies on the compressing property, which has been encoded in our random model. On the other hand, without this property, $\mathcal{L}_{aux}$ may include programs that never output the same on different inputs (e.g., the identity program $id\ x = x$), and it is easy to see that these programs are always valid for the weak specification. There is no guarantee that these programs must also satisfy the precise specification, and thus the introduced incompleteness may be out of control.

AutoLifter. Based on the above lemmas, we prove that the condition of the Occam-decomposition theorem is satisfied by each decomposition in *AutoLifter*, when both client synthesizers $\mathcal{S}_{aux}$ and $\mathcal{S}_{comb}$ are Occam solvers. Therefore, by applying this theorem, we prove that the probability for *AutoLifter* to fail on a solvable task can be arbitrarily small when a large enough input domain is considered, i.e., the probabilistic completeness of *AutoLifter* is ensured.

THEOREM 5.4 (PROBABILIC COMPLETENESS). *When client synthesizers $\mathcal{S}_{aux}$ and $\mathcal{S}_{comb}$ are both complete Occam solvers, the probability for AutoLifter to be incomplete, i.e., fail on a random solvable lifting problem under our model, converges to 0 as the size limit on the input domain becomes infinite.*

6 CLIENT SYNTHESIZERS

In this section, we discuss the implementation of the two client synthesizers. We assume that there are verifiers available for both subtasks generated by variable elimination and apply the CEGIS framework [Solar-Lezama et al. 2006] to convert the tasks into example-based tasks, i.e., synthesizing a program from a set of given examples.

6.1 Example-based solver in $\mathcal{S}_{aux}$

The task for $\mathcal{S}_{aux}$ is to synthesize an auxiliary program aux_1 from the first subtask generated by variable elimination (Formula 3). To see the example-based task here, let us first transform the specification into the following equivalent form.

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

For this specification, an example generated by CEGIS is an assignment to $(c, \overline{a}, \overline{a'})$. This assignment must satisfy the premise since it is generated as a counterexample of some candidate auxiliary program, and it requires aux_1 to satisfy the consequence, i.e., $aux_1^n\ \overline{a} \neq aux_1^n\ \overline{a'}$.

Example 6.1. Continuing to Example 4.2, the following shows two examples of task #3 in Figure 6, i.e., the first subtask generated by variable elimination from task #2.

Id	(xs_L, xs_R)	(xs'_L, xs'_R)	premise	requirement
e_1	$([1],[1])$	$([1],[-1,1])$	$(1,1)=(1,1) \wedge 2 \neq 1$	$(aux_1\ [1], aux_1\ [1]) \neq (aux_1\ [1], aux_1\ [-1,1])$
e_2	$([1],[1])$	$([1,-1],[1])$	$(1,1)=(1,1) \wedge 2 \neq 1$	$(aux_1\ [1], aux_1\ [1]) \neq (aux_1\ [1,-1], aux_1\ [1])$

The maximum prefix sum *mps* satisfies e_1 as mps^2 outputs differently on $([1],[1])$ and $([1],[-1,1])$. Similarly, the maximum tail sum *mts* satisfies e_2, and their pair $mts \triangle mps$ satisfies both examples.

The example-based task here is specified by a list of such examples, and its goal is to find a program in $\mathcal{L}_{aux}$ satisfying all examples. Because there is no specialized synthesizer for examples in this form, we implement the example-based solver in $\mathcal{S}_{aux}$ based on *observational equivalence (OE)* [Udupa et al. 2013], a general-purpose enumerative solver. OE enumerates programs by combining enumerated programs with language constructs and prunes off duplicated programs that output the same on all inputs involved by the given examples. Note that OE is a 1-Occam solver with constant 1 since it enumerates programs from small to large.

To improve the efficiency of OE, we also propose a specialized optimization, namely *observational covering*. We observe that in a lifting problem, a complex auxiliary program is usually formed as a tuple of simpler programs. For example, the full auxiliary program for applying D&C to *mss* is a tuple of *mps*, *mts* and *sum*. For a tuple-formed auxiliary program, it satisfies an example if and only if some component in it does. So, a program f should not be selected as a component in the auxiliary program if it is *covered*, that is, the set of examples f satisfies is covered by that of some other programs. At this time, all tuples including f can be safely ignored.

Example 6.2. Continuing to the previous example, program *max* that returns the maximum in a list satisfies neither e_1 nor e_2 and thus is covered by *mps*, which satisfies e_1. Therefore, in the sense of finding an auxiliary program satisfying both e_1 and e_2, tuples including *max* can be safely skipped. For any tuple $max \triangle f$ that satisfies e_1 and e_2, $mps \triangle f$ must also satisfy both examples.

We integrate observational covering into OE and meanwhile ensure that the resulting solver is still an Occam solver. In brief, we maintain the set of uncovered programs during the enumeration and use only these programs while constructing the tuple of the auxiliary program. More details on this example-based solver can be found in Appendix B.1.

6.2 Example-based solver in $\mathcal{S}_{comb}$

The task for $\mathcal{S}_{comb}$ is to synthesize a combinator corresponding to the synthesized aux_1 (from Figure 7b). Here, an example is an assignment to $(c, \overline{a})$, and it requires $comb_1$ to output $orig\ (op\ (c, \overline{a}))$ on input $(c, (aval \triangle aux_1)^n\ \overline{a})$. Such an example can be regarded as an input-output example for $comb_1$, and thus we use *PolyGen* [Ji et al. 2021], a state-of-the-art Occam solver based on input-output examples, as the example-based solver in $\mathcal{S}_{comb}$.

7 IMPLEMENTATION

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

Verification. The client synthesizers in *AutoLifter* use the CEGIS framework, and thus they require verifiers to verify the candidate programs synthesized from examples. In our implementation, we use a probabilistic verifier by default. It generates random examples from a pre-defined distribution on lists and integers and then tests the candidate program on these examples. Such a verifier does not access the syntax of the original program *orig* and thus keeps the generality of *AutoLifter*. By properly setting the number of examples, our verifier ensures that the probability for the error rate of the synthesized program to be more than 10^{-3} is at most 1.82×10^{-4}. More details on our verifier can be found in Appendix B.2.

Note that the verifier in *AutoLifter* can be replaced with any off-the-shelf verifier to get a stronger correctness guarantee. For example, if we assume all related programs are symbolically executable, the verifier can be replaced with bounded model checking [Biere et al. 2003].

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

- We take the DSL for list-related programs used by *DeepCoder* [Balog et al. 2017] as $\mathcal{L}_{aux}$.
- We take the DSL for conditional arithmetic in SyGuS-Comp [Alur et al. 2019] as $\mathcal{L}_{comb}$.

More details on these two DSLs used in *AutoLifter* can be found in Appendix B.3.

8 EVALUATION

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

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

8.1 Experimental Setup

Baseline Solvers. We compare *AutoLifter* with two inductive synthesizers, *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018]. Both synthesizers can be applied to the lifting problem and thus can be instantiated to inductive synthesizers for applying D&C-like paradigms as *AutoLifter* does.

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

We re-implement both *Enum* and *Relish* to support the list-related operations used in our paper.

We also compare *AutoLifter* with a state-of-the-art specialized synthesizer for D&C, namely *Parsynt* [Farzan and Nicolet 2017, 2021b]. *Parsynt* is a deductive synthesizer that requires the original program to be given as single-pass. It uses pre-defined rules to transform the loop body of *orig*, extracts auxiliary program *aux* directly, and then synthesizes a corresponding combinator *comb* via an inductive synthesizer. There are two versions of *Parsynt* available, where different transformation systems are used. We denote them as *Parsynt17* [Farzan and Nicolet 2017] and *Parsynt21* [Farzan and Nicolet 2021b] respectively, and consider both of them in our evaluation.

Dataset. Our evaluation is conducted on a dataset of 93 tasks of applying D&C-like algorithmic paradigms. They are related to four algorithmic problems and six paradigms.

Problem 1: applying D&C to a program. We collect 36 such tasks from the datasets used by previous studies [Bird 1989a; Farzan and Nicolet 2017, 2021b][5], including all tasks used by Bird [1989a]; Farzan and Nicolet [2017] and 12 out of 22 tasks used by Farzan and Nicolet [2021b]. The other 10 tasks used by Farzan and Nicolet [2021b] require a more general form of divide-and-conquer where the divide operator is not determined. Therefore, these tasks cannot be expressed by lifting problems and are out of the scope of *AutoLifter*.

[5]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.

Problem 2: applying single-pass to a program. Because there is no previous dataset on synthesizing single-pass programs, we construct this dataset based on the dataset of D&C. For each D&C task, we construct a single-pass task with the same original program. This task is useful in removing the requirement on the input program from existing deductive synthesizers for D&C. We can first apply *AutoLifter* to get a single-pass program and then use deductive synthesizers to generate a D&C program.

Problem 3: 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.

Problem 4: Longest Segment Problem. The longest segment problem is described by a predicate on lists and asks for the length of the longest segment in a given list satisfying the predicate. Zantema [1992] considers three different classes of predicates and proposes three corresponding paradigms. We consider the 8 sample tasks used by [Zantema 1992], including 3, 1, and 4 tasks corresponding to the three paradigms respectively. For each sample task, we include the task of applying the respective paradigm to it in our dataset.

All these tasks can be reduced to the lifting problem. We have illustrated the reductions for the first three problems in Section 3, and details on the last problem can be found in Appendix C. Through these reductions, *AutoLifter* and the baseline solvers *Enum* and *Relish* can be instantiated as synthesizers for applying these paradigms. As mentioned in Section 3, we ensure the efficiency of the synthesized algorithm by ensuring all combinators in $\mathcal{L}_{comb}$ are constant-time. At this time, we ensure (1) the D&C algorithm synthesized for Problem 1 runs in $O(n/p)$ time in parallel on $p \leq n/\log n$ processors, (2) the algorithm synthesized for Problem 2 and 4 runs in linear time, and (3) the algorithm synthesized for Problem 3 takes $O(\log n)$ time to respond to each query.

Besides, similar to previous studies on 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 thus directly apply the respectively instantiated synthesizer to each task in our evaluation. In practice, when there are multiple paradigms available, we may either invoke all corresponding synthesizers in parallel or deign a selector to select among them. This is out of the scope of this paper and is a direction for future work.

8.2 RQ1: Comparison of Synthesizers for Lifting Problems

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

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 8GB. To measure the efficiency of the solvers, we record the time cost for each successful synthesis.

Results. The results of this experiment are summarized in Table 1. For each setting and each solver, we report the number of tasks it solved in Column #Solved, its average time cost (seconds) on those solved tasks in Column T_{Base}, and the average time cost of *AutoLifter* on the same tasks in Column T_{Ours}. We conduct the following two manual analyses on the synthesis results.

Table 1. The results of Exp 1.

Solver	#Solved	T_{Base}	T_{Ours}
AutoLifter	**82**	6.66	
Enum	19	11.72	**0.14**
Relish	38	21.54	**5.33**

- As discussed in Section 7, our verifier provides only a probabilistic guarantee of correctness. Therefore, we manually verify all results and confirm that they are all **completely correct**[7].

[6] A common alias of segment trees while solving tasks of range update and range query.
[7] A gold medal winner in international programming competitions helped us to verify the synthesized algorithms.

Table 2. The results of comparing *AutoLifter* with *Parsynt*.

Solver	#Tasks	#$S_{\mathtt{Base}}$	#$S_{\mathtt{Ours}}$	$T_{\mathtt{Base}}$	$T_{\mathtt{Ours}}$	#$\mathrm{Aux}_{\mathtt{SP}}$
ParSynt17	20	**19**	**19**	15.59	**5.84**	39.29%
ParSynt21	36	24	**29**	6.86	**4.19**	56.90%

- For each execution of *AutoLifter*, we manually verify all usages of our decomposition methods and confirm that the program synthesized from the weak specification never goes wrong. This result matches our probabilistic guarantee of completeness (Property 5.4).

Compared with existing synthesizers, *AutoLifter* achieves significantly better performance. It not only solves much more benchmarks but also solves faster on those jointly solved tasks.

AutoLifter fails in solving 11 out of 93 tasks in our dataset. These failures are all because of the limited expressiveness of the default DSLs used in our implementation. To solve these tasks, specialized operators such as regex matching on an integer list and the power operator on integers are required, which are not included in the general-purpose DSLs we used. After manually providing these missing operators, *AutoLifter* can solve 10 more tasks and find a valid auxiliary program on the last remaining task[8]. Such a result shows that *AutoLifter* can be further improved if missing operators can be inferred automatically. To achieve this, one possible way is to incorporate those deductive approaches and then extract those useful operators from the user-provided implementation. This is a direction for future work.

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

Procedure. We compare *AutoLifter* with the two versions of *Parsynt* on tasks for D&C in our dataset. Because *ParSynt* requires the original program to be single-pass, we provide such an implementation for each task to invoke *ParSynt*[9]. Note that such a comparison favors *ParSynt* because some required auxiliary values are directly provided to *ParSynt* in the single-pass implementation.

We failed in installing *ParSynt17* because of some issues with the dependencies. The authors of *ParSynt17* confirmed but have not solved this problem. So we compare *AutoLifter* with *Parsynt17* only on its original dataset using the evaluation results reported by Farzan and Nicolet [2017].

Similar to Section 8.2, we use a time limit of 300 seconds, use a memory limit of 8GB, and record the time cost for each successful synthesis.

Results. The results of this experiment are summarized in Table 2. We report the number of tasks considered in the comparison in column #Tasks, the numbers of tasks solved by *Parsynt* and *AutoLifter* in column #$S_{\mathtt{Base}}$ and #$S_{\mathtt{Ours}}$, the average time cost (seconds) of the two solvers in column #$T_{\mathtt{Base}}$ and #$T_{\mathtt{Ours}}$, and the ratio of the number of auxiliary values provided in the provided single-pass algorithm to the number of auxiliary values used in the divide-and-conquer algorithm synthesized by *Parsynt* in column #$\mathrm{Aux}_{\mathtt{SP}}$%. We consider only those tasks solved by both *Parsynt* and *AutoLifter* while calculating the average time cost and the ratio.

Please note that both *AutoLifter* and the two versions of *Parsynt* ensure the time complexity of the synthesized algorithm to be exactly $\Theta(n/p)$ on $p \leq n/\log n$ processors when a parallel template is used. Therefore, once successfully synthesized, the time complexity of the synthesized programs would be the same.

[8]This last failed task is *longest_odd_(0+1)* constructed by Farzan and Nicolet [2021b]. On this task, *AutoLifter* fails because the client synthesizer *PolyGen* times out in finding a corresponding combinator.

[9]For those tasks taken from *Parsynt*, we use the program in its original evaluation and fix the two bugs we found.

The results show that, though about $40\% - 60\%$ auxiliary values and the syntax information are provided to *Parsynt*, *AutoLifter* still achieves competitive performance compared with *Parsynt* on synthesizing divide-and-conquer algorithms.

Now, we would like to report two results we observed while analyzing the synthesis results.

(1) *AutoLifter* never uses more auxiliary values than *ParSynt* and uses strictly fewer on 10 tasks in total. This is because the syntax information may mislead *Parsynt* to some unnecessarily complex solutions. Let us take task *line_sight* (*ls*) as an example, which checks whether the last element is the maximum in a list. It can be implemented as single-pass with auxiliary program *max*, which returns the maximum of a list, since $ls\ (xs \mathbin{+\!\!+} [v]) = v \geq (max\ xs)$. Given this program, *Parsynt* will extract *last l* as an auxiliary value because the last visited element v is directly used in the comparison. However, this value is not necessary because $ls\ (l_1 \mathbin{+\!\!+} l_2)$ is always equal to $(ls\ l_2) \wedge (max\ l_1 \leq max\ l_2)$. *AutoLifter* can find this simpler solution as it synthesizes directly from the semantics.

(2) For applying D&C, the issue of missing operators on *AutoLifter* can be alleviated by combining *AutoLifter* with *Parsynt*. Though the default DSLs are not expressive enough for applying D&C on 7 tasks, it is enough for applying single-pass on 5 among them. *AutoLifter* can successfully synthesize single-pass algorithms for these 5 tasks, and then *Parsynt21* can synthesize D&C algorithms for them. In this way, the combination of *AutoLifter* and *Parsynt* can solve 34 out of 36 tasks, outperforming both individual solvers.

8.4 Case Study

We also conduct a case study on two tasks in our dataset, showing that (1) the advantage of inductive synthesis, and (2) *AutoLifter* can solve tasks difficult for human programmers. Due to the space limit, we report the case study in Appendix D.

9 RELATED WORK

Automatic Applications of D&C-like Paradigms. In this paper, we consider a broad class of D&C-like paradigms, and thus our work is related to previous studies on applying such 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 divide-and-conquer, which can be reduced to lifting problems as discussed in Section 2. All of these approaches are deductive and require the input program to be implemented as single-pass. Compared with them, *AutoLifter* is the first inductive synthesizer that does not require single-pass implementations, and as shown in our evaluation, *AutoLifter* achieves competitive performance with state-of-the-art deductive synthesizers for divide-and-conquer.

Second, Acar et al. [2005] proposes a deductive approach for incrementalization. As discussed in Section 2.2, it constructs the auxiliary program and the combinator directly from the original program, and thus the efficiency of the synthesized algorithm greatly depends on how the original program is implemented. Compared with this approach, *AutoLifter* does not rely on the syntax and thus can usually find efficient algorithms even if the original program is not desirable. On the other hand, when a proper original program is given, Acar et al. [2005]'s approach can generate incrementalization algorithm for extremely difficult tasks such as generating dynamic data structures that require hundreds 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, there exist multiple synthesizers for divide-and-conquer and other D&C-like paradigms that do not support the synthesis of the auxiliary program, including divide-and-conquer [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 in the paradigm. Compared with these approaches, *AutoLifter* not only supports more paradigms but can also automatically find proper auxiliary values when necessary.

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 the approach proposed by Farzan and Nicolet [2021b] requires a single-pass implementation and the approach proposed by Miltner et al. [2022] 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 achieve more refined guarantees on the efficiency via type systems. However, these approaches 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 could scale up to synthesizing efficient D&C-like algorithms as our approach does.

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 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 based on Boolean and arithmetic operators, and thus are useless for lifting problems, where these operators are not explicitly used in the specification.

AutoLifter is also related to *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018] because these approaches are applicable to the lifting problem. They are general-purposed synthesizers and are applicable to lifting problems. We compare *AutoLifter* with both of them in our evaluation, and the results demonstrate the effectiveness of *AutoLifter*.

10 CONCLUSION

In this paper, we study the inductive synthesis problem for applying D&C-like algorithmic paradigms. We capture the application of various paradigms as lifting problems and propose a novel synthesizer *AutoLifter* for them. To solve lifting problems efficiently, we use two decomposition methods, component elimination and variable elimination, to divide a lifting problem into simpler subtasks and replace precise but complex specifications with weak but simple specifications. In theory, we prove the Occam-decomposition property to bound the incompleteness introduced by the weak specification, and in practice, we demonstrate the effectiveness of *AutoLifter* via an evaluation conducted on 4 algorithmic problems, 6 paradigms, and 93 lifting problems.

Though this paper focuses on lifting problems, many techniques proposed in *AutoLifter* are general and can be potentially applied to other tasks. For example, the general form of the decomposition method and the Occam-decomposition theorem can be used to decompose other complex tasks, and the method of variable elimination may be used to separate the composition of two unknown programs in other synthesis tasks. Exploring these applications is future work.

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. 1977. Solution to Klee"s Rectangle Problem. (1977).

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

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

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

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

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

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

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

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, 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