Black-Box Algorithm Synthesis

Divide-and-Conquer and More

Anonymous Author(s)

Abstract

Algorithm synthesis is a newly emerging branch of program synthesis, targeting to automatically apply a predefined class of algorithms to a user-provided program. In algorithm synthesis, one popular topic is to synthesize divide-and-conquer-style parallel programs. Existing approaches on this topic rely on the syntax of the user-provided program and require it to follow a specific format, namely *single-pass*. In many cases, implementing such a program is still difficult for the user. Therefore, in this paper, we study the black-box synthesis for divide-and-conquer which removes the requirement on the syntax and propose a novel algorithm synthesizer *AutoLifter*. Besides, we show that *AutoLifter* can be generalized to other algorithms beyond divide-and-conquer. We propose a novel type of synthesis tasks, namely *lifting problems*, and show that *AutoLifter* can be applied to those algorithms where the synthesis task is an instance of lifting problems. To our knowledge, *AutoLifter* is the first algorithm synthesizer that generalizes across algorithm types. We evaluate *AutoLifter* on two datasets containing 57 tasks covering five different algorithms. The results demonstrate the effectiveness of *AutoLifter* for solving lifting problems and show that though *AutoLifter* does not access the syntax of the user-provided program, it still achieves competitive performance compared with white-box approaches for divide-and-conquer.

1 Introduction

Algorithm synthesis is a newly emerging branch of program synthesis. An algorithm synthesizer targets a predefined class of algorithms, such as divide-and-conquer [Farzan and Nicolet 2017, 2021; Morita et al. 2007], dynamic programming [Lin et al. 2019], and incrementalization [Acar et al. 2005], and its task is to automatically apply these algorithms to a user-provided program. Algorithm synthesis imposes significant challenges for synthesizers because an algorithm often involves complex control structures and is usually large.

The most popular topic on algorithm synthesis is to automatically synthesize divide-and-conquer-style parallel programs for lists. Given a target function that takes a list as input and produces a value as output, a divide-and-conquer algorithm splits the input list into two sublists, recurses into them in parallel, and at last merges the result together via a combinator. In general, the outputs of the target function on the sublists may not be enough for calculating the output of the function on the whole list. For these functions, the algorithm synthesizer needs to find proper functions, namely *lifting functions*, which produce supplementary *lifting values* on the sublists such that the output of the target function on the whole list can be calculated.

To find lifting functions, existing approaches [Farzan and Nicolet 2017, 2021; Fedyukovich et al. 2017; Raychev et al. 2015] require the original program to be *single-pass*. A single-pass program on lists is an instance of $\lambda l : [E].(\mathit{fold}\ ?\oplus\ ?e\ l)$, where $?e : D$ is an initial state and $?\oplus : D \times E \to D$ is a function that updates the state with an element in the list. The single pass program visits all elements in the input list l in order, and updates the state via $?\oplus$. Existing approaches find the lifting functions via deductive methods. They use pre-defined rules to transform $?\oplus$ and $?e$, and either extract lifting functions directly or decompose the synthesis task into simpler subtasks by analyzing the input program. As a result, for these approaches, the efficiency of the synthesized program depends on the input program, and the input program has to be efficient to obtain an efficient result.

However, in many cases, providing a single-pass implementation can be difficult for the user. On the one hand, an efficient single-pass implementation still requires the user to find proper lifting functions. As we shall show in Section 7.3, on a dataset collected from previous work, the number of lifting values required to write an efficient single-pass program accounts for 41.1%-60.6% of the number of lifting values required to directly write a divide-and-conquer-style parallel program. On the other hand, implementing a single-pass function is error-prone even for experts. The dataset used by Farzan and Nicolet [2021] contains two bugs that were introduced when the authors manually rewrote the original program into a single-pass program[1].

To further reduce the burden on the user, we study a more general synthesis task by removing the requirement on the syntax of the original program. We name this problem as *black-box algorithm synthesis* for divide-and-conquer.

Following previous studies on program calculation [Bird and de Moor 1997], we use *algorithmic tactics* to simplify the synthesis task. *Program calculation* techniques aim to establish an algebra for human users to derive programs. In this domain, an *algorithmic tactic* summarizes a class of algorithms as an *algorithmic template* with variables representing task-related program fragments and an *application condition* for these variables to form a correct program. We

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

[1] See the footnote on page 10 for details.

obtain a program by filling the algorithmic template with a proper assignment to the variables satisfying the application condition. Because the main control structure is usually captured by the template, the assignment is usually simpler than the result in both the control structure and the scale.

Guided by the tactic for divide-and-conquer-style parallelization [Cole 1995], the task of synthesizing divide-and-conquer programs is converted to automatically synthesize an assignment to the variables satisfying the corresponding application condition. Though the conversion has greatly simplified the task of algorithm synthesis, due to the black-box setting of our approach, there are still significant challenges on both *synthesis* and *verification*.

For synthesis, the assignment to variables can still be large. In our evaluation, the program in the assignment has up to 157 AST nodes. In contrast, applicable synthesis techniques are limited. First, the deductive methods used in previous work become unavailable because the original program is no longer guaranteed to be single-pass. Second, most state-of-the-art inductive methods, such as λ^2 [Feser et al. 2015] and witness functions [Polozov and Gulwani 2015], cannot be used because (1) the application condition includes the composition of two variables, and (2) input-output examples of one variable cannot be extracted from the condition.

For verification, most synthesizers rely on a verifier to determine the correctness of the synthesis result. However, existing verification techniques hardly scale up to synthesizing divide-and-conquer because the verification involves data structures and has no assumption on the syntax.

The first contribution of our paper is an efficient synthesizer, namely *AutoLifter*, for solving the application condition corresponding to divide-and-conquer.

For synthesis, *AutoLifter* combines deductive methods and inductive methods. In the deductive part, *AutoLifter* includes two novel deductive rules, namely *decomposition* and *decoupling*, which are based on the structure of the application condition only. The deductive part generates several subtasks, each synthesizing for a part of a single functional variable. Then, the inductive part solves these tasks by either *PolyGen* [Ji et al. 2021], a state-of-the-art synthesizer for input-output examples, or *observational covering*, a novel enumerative strategy we propose in this paper.

For verification, due to the difficulty of modeling, *AutoLifter* verifies in a probabilistic way. We show that the inductive solvers used by *AutoLifter* are all Occam solvers [Ji et al. 2021]. By combining Occam solvers and an iteratively increasing number of examples, we ensure *AutoLifter* has a configurable probabilistic guarantee on the correctness.

The second contribution of this paper is to generalize *AutoLifter* to other algorithmic tactics beyond divide-and-conquer. Since *AutoLifter* uses only the structure of the application condition and has no requirement on the syntax of the input program, *AutoLifter* can be naturally generalized to synthesize for all algorithmic tactics whose application condition has a similar structure to divide-and-conquer. Such tactics cover different types of algorithms for different types of problems, such as greedy algorithms for longest segment problems [Zantema 1992], dynamic programming algorithms for maximum weightsum problems [Sasano et al. 2000], a novel data structure for Klee's rectangle problems [Bentley 1977], etc. We define these tasks uniformly as *lifting problems* and generalize *AutoLifter* to all lifting problems. To our knowledge, no existing approach on algorithm synthesis has such ability to generalize across algorithm types.

The third contribution of this paper is a set of experiments evaluating the performance of *AutoLifter*. First, to evaluate the effectiveness of *AutoLifter* on synthesizing divide-and-conqer algorithms, we collect a dataset of 36 tasks from previous work [Bird 1989; Farzan and Nicolet 2017, 2021; Morita et al. 2007]. We compare *AutoLifter* with a state-of-the-art while-box solver, *Parsynt* [Farzan and Nicolet 2017, 2021], on this dataset. The results show that though *AutoLifter* does not enforce specific requirements on the syntax of the original program, it can still achieve competitive, or even better, performance compared with white-box solvers. Second, to evaluate the effectiveness of *AutoLifter* on other tactics, we collect a dataset of 22 tasks from an existing publication on program calculation [Zantema 1992] and an online contest platform for competitive programming (codeforces.com), covering 4 other algorithmic tactics. The results show that *AutoLifter* is able to solve all of these tasks with an average time cost of 7.52 seconds. At last, we establish a case study on two tasks in our dataset, which demonstrates *AutoLifter* (1) can find results that are counterintuitive on syntax, and (2) can solve problems that are hard even for world-level players in competitive programming.

To sum up, this paper explores the problem of *black-box algorithm synthesis* and makes the following contributions.

- Proposing an efficient black-box synthesizer, *AutoLifter*, for synthesizing divide-and-conquer programs. (Section 4)
- Defining the *lifting problem* that captures the synthesis tasks for multiple algorithmic tactics and generalizing *AutoLifter* to lifting problems. (Section 5)
- Conducting an evaluation on two datasets and 57 tasks, and showing the effectiveness of *AutoLifter*. (Sections 6, 7)

2 Algorithmic Tactic for D&C

We introduce the algorithmic tactic for divide-and-conquer via a classical problem, *maximum segment sum (mss)*:

Given list l, the task is to select a segment (i.e., consecutive subsequence) s from l and maximize the sum of elements in s.

Function p at line 3 in Figure 1 implements an exhaustive search for *mss* in Haskell. This algorithm enumerates all segments of the original list, calculates their sums, and returns the maximum one. Concretely, `inits` returns all prefixes of a list, `tails` returns all suffixes of a list, and thus `t` enumerates over all suffixes of all prefixes, i.e., all segments.

```
1  import Data.List
2  -- Original program
3  p l = maximum (map sum
              [t | i <- inits l, t <- tails i])
4  -- Variables
5  f l = (mps l, mts l, sum l)
6    where
7    mts = maximum.(scanr (+) 0)
8    mps = maximum.(scanl (+) 0)
9  c (mss1, (mps1, mts1, sum1))
10   (mss2, (mps2, mts2, sum2)) =
                    (mss3, (mps3, mts3, sum3))
11   where
12   mss3 = maximum [mss1, mss2, mts1 + mps2]
13   mps3 = max mps1 (sum1 + mps2)
14   mts3 = max (mts1 + sum2) mts2
15   sum3 = sum1 + sum2
16 -- Algorithmic template
17 dc' l =
18   if length l <= 1 then (p l, f l)
19   else c (dc' (take m l)) (dc' (drop m l))
20   where m = div (length l) 2
21 dc = fst.dc'
```

Figure 1. A divide-and-conquer program for *mss*.

The exhaustive search algorithm is inefficient, as its time complexity is $O(n^3)$, where n is the length of the input list. To optimize the performance of the algorithm, the rest of the code in Figure 1 shows an efficient divide-and-conquer-style parallel program, implemented as function dc (line 21). The basic idea of the algorithm is to divide the list into two halves, recursively apply itself to the two halves, and use a combinator to obtain the maximum segment sum of the original list from the results of the two halves (Line 19).

Because it is not enough to calculate the maximum segment sum of the whole list from those of the two halves, the algorithm uses a function f to calculate supplementary information needed for the combinator (Line 19). In this case, the supplementary information includes the maximum tail sum (*mts*) and maximum prefix sum (*mps*), as the maximum segment could be formed by a tail of the first half and a prefix of the second half. To further calculate *mts* and *mps* of the whole list, another value, the element sum is also calculated (lines 5-8). In this paper, we name *mts*, *mps*, *sum* as *lifting functions*, whose result is used during divide-and-conquer, and name f as a *lifting scheme*, which summarizes all necessary lifting functions in a tuple.

Based lifting scheme f, function c calculates the maximum segment sum as well as the outputs of the three involved lifting functions by combining those of the two halves (lines 9-15). Since the two invocations to dc' (Line 19) can be executed in parallel, the time complexity of this algorithm is $O(n/t)$ when $t = O(n/\log n)$ processors are given.

```
1  import Data.List
2  e = (0, 0)
3  oplus (mss1, mts1) x =
         (max mss1 (mts1 + x), max 0 (mts1 + x))
4  sp l = fst (foldl oplus e l)
```

Figure 2. A single-pass program for *mss*.

This program also shows the algorithmic tactic for divide-and-conquer [Cole 1995], which guides the user to rewrite a program p as a divide-and-conquer-style parallel program.

- First, the user needs to find two auxiliary functions f and c. Function f is a lifting scheme that calculates the outputs of lifting functions needed for the divide-and-conquer, and c is a *combinator* that calculates values of p and f from the results of the recursive invocations. For correctness, f and c should satisfy the following formula for all lists l_1, l_2.

$$(p\ (l_1 +\!\!+ l_2), f\ (l_1 +\!\!+ l_2)) = c\ \Big((p\ l_1, f\ l_1), (p\ l_2, f\ l_2)\Big) \quad (1)$$

  where $l_1 +\!\!+ l_2$ represents the concatenation of l_1 and l_2.
- Second, the user needs to fill programs p, f, and c to a template, which has been shown in lines 17-21 of Figure 1.

That is, with an algorithmic tactic, the user only needs to find the auxiliary functions satisfying the application condition but does not need to know the details of the algorithm.

Formally, an algorithmic tactic is a pair $\mathcal{A} = (\varphi, T)$:

- Application condition φ is a formula with respect to the original program p and several variables $g_1, \ldots, g_n$. To apply the algorithmic tactic, the user is required to find a valid assignment for $g_1, \ldots, g_n$.
- Algorithmic template T is a partial program with some holes remaining. T can be completed by filling these holes with p and a valid assignment for $g_1, \ldots, g_n$.

3 Overview

In this paper, we introduce the main idea of *AutoLifter* using the *mss* problem mentioned in Section 2. The original program and one target program of this task have been shown as program p and dc (Lines 5-21) in Figure 1 respectively.

3.1 Shortage of White-Box Approaches

To synthesize program dc, one major challenge is to find the lifting functions, i.e., *mps*, *mts*, and *sum*. Some white-box approaches [Farzan and Nicolet 2017, 2021; Fedyukovich et al. 2017; Raychev et al. 2015] are able to find these functions. However, they require the original program to be single-pass, and thus cannot be directly applied to program p. Moreover, they also rely on the efficiency of the original program. To synthesize a program as efficient as dc, they require the original program to be linear-time.

Figure 2 shows a valid input sp for these approaches. Starting from the initial value e (Line 2), sp scans the input list l

from left to right and updates the result via the loop body `oplus` after visiting each element `x` (Line 3). Similar to `dc`, since *mss* itself is not enough to obtain the next *mss* during the loop, `sp` uses lifting function *mts*.

Compared with `p` (Line 3 in Figure 1), `sp` (Lines 2-4 in Figure 2) is a much more difficult for the user to implement. On the one hand, to ensure efficient single-pass implementation, the user has to find the lifting value *mts*. It is actually a similar task with finding lifting functions for divide-and-conquer. On the other hand, the user needs to capture the change of *mts* and *mps* during the loop. For example, the user must recognize that *mts* is at least 0 during the loop. Such a task is sometimes error-prone.

Therefore, though these white-box approaches synthesize a divide-and-conquer program from a single-pass program, there is still a significant burden remaining to the user.

3.2 Overview on *AutoLifter*

Different from existing white-box approaches, *AutoLifter* makes no assumption on the syntax of the original program. Therefore, program `p` is a valid input for *AutoLifter*. By the algorithmic tactic discussed in Section 2, the algorithm synthesis task is simplified to finding functions f and c such that Equation 1 is satisfied for all lists l_1, l_2. Figure 3 shows the procedure for *AutoLifter* to solve this task and some results for the *mss* task, where p is equal to `p` (Line 1 in Figure 1).

Deductive Part. The first challenge is on the scale of the target programs. As shown in Figure 3, the target f and c use 7 and 21 operators respectively, which are far beyond the scope of state-of-the-art synthesizers for list-operating programs. For example, *DeepCoder* [Balog et al. 2017], a state-of-the-art synthesizer on lists, times out on $\geq 40\%$ tasks in a dataset for synthesizing list-operating programs with 5 operators, even when the time limit is one hour.

AutoLifter uses two deductive rules to solve this challenge. Given a synthesis task, the deductive rules split it into subtasks and merge the results of subtasks into a solution to the original task. In each subtask, only a part of a single target program is synthesized, and thus the scale is reduced.

The first rule *decomposition* splits the original task (Task 1 in Figure 3) according to the usage of lifting functions. In our example, there are 3 lifting functions *mps*, *mts* and *sum*. They can be divided into two lifting schemes f_1 and f_2.

- (f_1) The first scheme provides supplementary information for the input program p, including *mps* and *mts*.
- (f_2) The second scheme provides supplementary information for calculating other lifting values, including *sum*.

Rule *decomposition* generates two subtasks (Task 2 and Task 5 in Figure 3) for synthesizing f_1 and f_2 respectively.

- Group f_1 provides supplementary information only for p, so we remove the second component $f\ (l_1 +\!\!+ l_2)$ on the left-hand side of Task 1, resulting in Task 2.
- Group f_2 provides supplementary information for calculating f_1, and its logic specification is shown as Task 5. Because f_1 has already provided the supplementary information for p, p is no longer considered at the left-hand side. But p still occurs at the right-hand side as its output can be used to calculate the outputs of lifting functions.

Note that Task 5 has the same form as Task 1, and thus can be solved by applying *decomposition* recursively.

The second rule *decoupling* decouples the composition of variables at the right-hand side of Task 2 (and other similar subtasks). Starting from Task 2, *decoupling* first extracts the specification for f_1 by requiring the existence of c.

$$\exists c_1, \forall l_1, l_2, p\ (l_1 +\!\!+ l_2) = c_1\ ((p\ l_1, f_1\ l_1), (p\ l_2, f_1\ l_2))$$

Since c_1 is a function, i.e., the same input always leads to the same output, the above specification is equivalent to the formula of Task 3. Note that such an equivalency depends on the fact that c can be any function. In practice, c is always constrained by grammar, and thus Task 5 is only a necessary condition. We prove that when the grammar satisfies some properties, the effectiveness of *decoupling* is still guaranteed. More details on this point can be found in Section 4.2.

After finding f_1, the corresponding c_1 can be synthesized by substituting f_1 into Task 2, resulting in Task 4.

Inductive Part. After the deductive system, there are two types of subtasks remaining. The first type is for a part of the lifting scheme f (e.g., Task 3) and the second type is for a part of the combinator c (e.g., Task 4). *AutoLifter* uses inductive methods to solve these subtasks.

We start the discussion from the second type of subtasks. There are two noticeable properties in Task 4:

- Input-output examples of c_1 can be extracted from Task 4. For any two lists l_1, l_2, the input and the output of c_1 are $((p\ l_1, f_1\ l_2), (p\ l_2, f_1\ l_2))$ and $p\ (l_1 +\!\!+ l_2)$ respectively.
- Though Task 4 is list-related, input-output examples of c_1 involve only the outputs of the original program and lifting functions, which are usually scalars in practice.

Therefore, *AutoLifter* uses *PolyGen* [Ji et al. 2021] to solve Task 4 as well as all subtasks in the second type. *PolyGen* is a state-of-the-art synthesizer for *conditional linear integer arithmetic* and is based on input-output examples.

In contrast, extracting input-output examples is difficult for the first type of subtasks such as Task 3. The formula of Task 3 is a metamorphic relation, i.e., a necessary condition of f over multiple inputs, and thus does not give a unique output given an input. Therefore, *AutoLifter* solves the first type of subtasks based on *observational equivalence* [Udupa et al. 2013], a state-of-the-art algorithm that does not rely on input-output examples. To improve *observational equivalence*, we further propose a novel pruning strategy, *observational covering*, on top of observational equivalence.

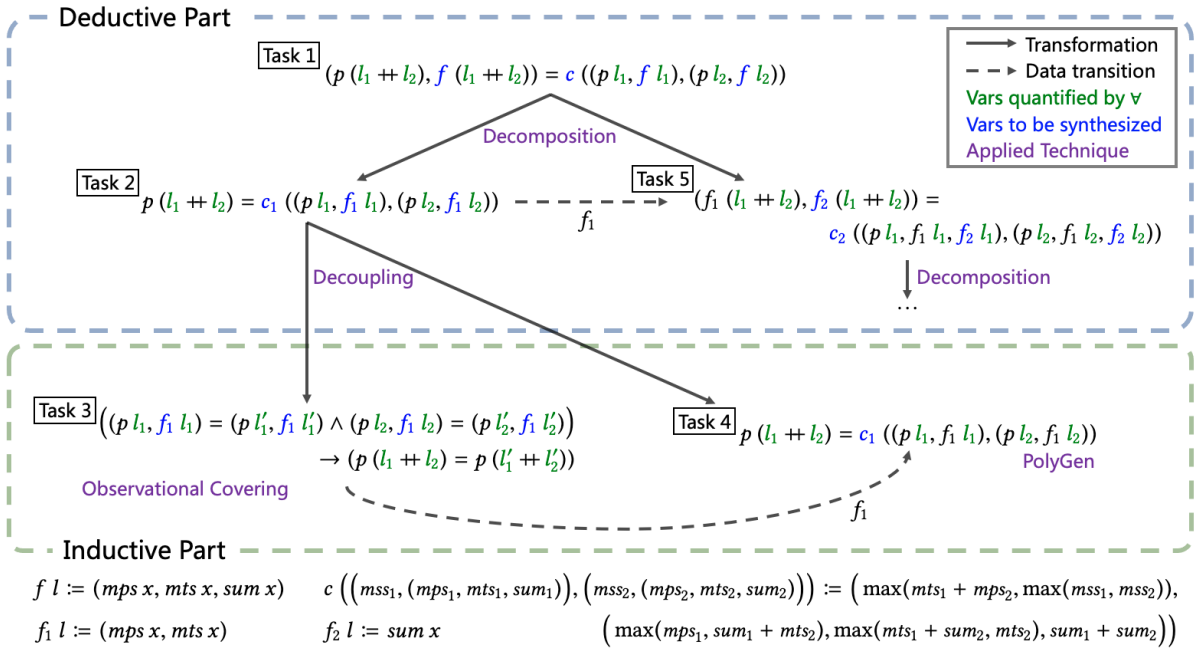

Figure 3. The procedure for *AutoLifter* to synthesize a divide-and-conquer program for original program p. We list the abbreviations of some synthesis results for the *mss* task at the bottom, where $mps\ l$, $mts\ l$, $\max(x, y)$ are abbreviations for $max\ (scanl\ (+)\ l)$, $max\ (scanr\ (+)\ l)$ and $ite\ (x < y)\ x\ y$ respectively. More details on the grammars can be found in Section 6.

We first introduce how *observational equivalence* works. It starts from atomic programs, and repeatedly combines enumerated programs to form larger programs until a valid program is found. During this procedure, if a smaller program e outputs the same as a larger program e' on a set of predetermined input examples, i.e., observationally equivalent, e' will be pruned off from the set of enumerated programs.

Observational covering shares the same idea of pruning off larger programs that do not contribute more than a smaller program with respect to a set of pre-determined examples, and further utilizes the fact that a lifting scheme is a tuple of lifting functions. By transforming the formula of Task 3, we obtain the following equivalent formula.

$$\left(\wedge_{i\in\{1,2\}}\ (p\ l_i = p\ l_i')\wedge p\ (l_1 +\!\!+ l_2) \neq p\ (l_1' +\!\!+ l_2')\right) \longrightarrow \vee_{i\in\{1,2\}}(f_1\ l_i \neq f_1\ l_i')$$

That is, when the input lists satisfy some condition, lifting scheme f_1 should return different results on some pair of input lists. Because f_1 is a tuple of lifting functions, f_1 returns different results when any of the lifting functions return different results. Therefore, a lifting function is not useful if the set of examples it satisfied in the above formula is *covered* by an existing lifting function.

Based on the above analysis, if there are two enumerated programs e, e' satisfying (1) e is smaller than e' and (2) e satisfies all examples satisfied by e', e' will not be considered as a lifting function to form a lifting scheme. For example, when the example is $([1], [1], [-1, 1], [1])$, program $\lambda l.(max\ l + 1)$ will be pruned off by $\lambda l.max\ l$.

Efficiency of the Result. One important detail is that not all solutions to Task 1 can lead to an efficient divide-and-conquer program. For example, it is easy to verify that $f\ l := l$ and $c\ ((\_, l_1), (\_, l_2)) := mss\ (l_1 +\!\!+ l_2)$ form a solution to Task 1. However, after filling them into the template, the time complexity of the result is still $O(n^3)$.

To guarantee the efficiency of the result, *AutoLifter* uses *syntax-guided synthesis* [Alur et al. 2013]. *AutoLifter* is configured by two grammars G_f and G_c. While synthesizing f and c, only programs in G_f and G_c are considered, respectively.

It can be proven that the time complexity of the result is guaranteed to be $O(n/t)$ on $t = O(n/\log n)$ processors when the combinator c runs in constant-time. *AutoLifter* guarantees this point by making **two assumptions**.

1. Operators on scalar values in G_c and G_f are all constant time (e.g., $+$, $-$, *and*, *or* and the branch operator *ite*).
2. Original program p and programs in G_f always return a constant number of scalar values.

Through these assumptions, scheme $f\ l := l$ is excluded, as it returns a list instead of a constant number of scalar values.

Verification. Both of the two inductive synthesizers require a verifier to determine the correctness of the result. However, it is difficult to use off-the-shelf verifiers in either Task

3 or Task 4, because *mss* is complex. To address this problem, *AutoLifter* verifies in a probabilistic way and provides a configurable probabilistic guarantee on the correctness.

The verification is based on *Occam solvers* [Ji et al. 2021]. An important property of Occam solvers is that, when the number of examples is larger than a polynomial to the size of the smallest valid program, the program synthesized by an Occam solver has a probabilistic guarantee on correctness [Blumer et al. 1987]. We prove that the inductive solvers used by *AutoLifter* are Occam solvers, and thus use this property to build the verifier. The main challenge here is to determine a proper number of examples provided to the solver, as the size of the smallest valid program is unknown.

AutoLifter achieves this by iterating on a parameter t. In each turn, *AutoLifter* assumes the size of the target program is at most t and chooses a proper number of examples according to the guarantee provided by Occam solvers. When the size of the synthesized program is no larger than t, the size of the smallest valid program must be no larger than t, and thus this result can be safely returned. Otherwise, t will be doubled, and a new turn will start. In this way, the correctness of the synthesis result is guaranteed.

4 Approach for Divide-and-Conquer

In this section, we give the details on how *AutoLifter* synthesizes divide-and-conquer programs. Given original program p and two grammars G_f, G_c, the task for *AutoLifter* is to find a lifting scheme f from G_f and a combinator c from G_c satisfying the following formula for all lists l_1, l_2.

$$(p\ (l_1 +\!\!+\ l_2), f\ (l_1 +\!\!+\ l_2)) = c\ ((p\ l_1, f\ l_1), (p\ l_2, f\ l_2))$$

Specially, f can be constant function *null*, representing that no supplementary information is required. To ensure the efficiency of the result, *AutoLifter* requires p and G_f, G_c to satisfy the two assumptions introduced in Section 3.

For convenience, we use $\circ$, $\triangle$ and $\times$ to represent the composition and the product of functions, where $(f \circ g) := f\ (g\ x)$, $(f \triangle g)\ x := (f\ x, g\ x)$, and $(f \times g)\ (x_1, x_2) := (f\ x_1, g\ x_2)$.

4.1 Subtasks

There are four types of subtasks generated during the synthesis procedure of *AutoLifter*. Similar to Figure 3, we color those variables quantified by quantifier $\forall$ as green and color variables corresponding to the synthesis targets as blue.

- Lifting problem $\mathsf{LP}(p, h)$ for synthesizing f and c.

$$(p\ (l_1 +\!\!+\ l_2), f\ (l_1 +\!\!+\ l_2)) = c\ ((h\ l_1, f\ l_1), (h\ l_2, f\ l_2))$$

- Partial lifting problem $\mathsf{PLP}(p, h)$ for synthesizing f and c.

$$p\ (l_1 +\!\!+\ l_2) = c\ ((h\ l_1, f\ l_1), (h\ l_2, f\ l_2))$$

- Subtask $\mathsf{S}_f(p, h)$ for synthesizing f.

$$\left(\wedge_{i\in\{1,2\}}\ (h\ l_i = h\ l_i') \wedge p\ (l_1 +\!\!+\ l_2) \neq p\ (l_1' +\!\!+\ l_2')\right) \longrightarrow \vee_{i\in\{1,2\}}(f\ l_i \neq f\ l_i')$$

- Subtask $\mathsf{S}_c(p, h, f)$ for synthesizing c.

$$p\ (l_1 +\!\!+\ l_2) = c\ ((h\ l_1, f\ l_1), (h\ l_2, f\ l_2))$$

Example 4.1. The types of the 5 tasks shown in Figure 3 are listed in the following table. To avoid confusion with the generic original program p here, we refer to the original program in our example as *mss*.

Task 1	$\mathsf{LP}(mss, mss)$	Task 2	$\mathsf{PLP}(mss, mss)$
Task 3	$\mathsf{S}_f(mss, mss)$	Task 4	$\mathsf{S}_c(mss, mss, mps\triangle mts)$
Task 5	$\mathsf{LP}(mss, mss\triangle(mps\triangle mts))$		

Note that in this section, we focus on the tactic of divide-and-conquer only. The concepts of these subtasks will be generalized and redefined in Section 5.

4.2 Deductive Part

The deductive system of *AutoLifter* splits a lifting problem $\mathsf{LP}(p, h)$ into several subtasks of type S_f and S_c via two deductive rules, *decomposition* and *decoupling*.

Decomposition. Given lifting problem $\mathsf{LP}(p, h)$, the procedure of *decomposition* is listed below.

1. Solve subtask $\mathsf{PLP}(p, h)$. Let (f_1, c_1) be the result.
2. Return $(f_1, c_1 \triangle null)$ when $f_1 = null$.
3. Solve subtask $\mathsf{LP}(f_1, h \triangle f_1)$. Let (f_2, c_2) be the result.
4. Return $\left(f_1 \triangle f_2, (c_1 \circ (\varphi_l \times \varphi_l)) \triangle (c_2 \circ (\varphi_r \times \varphi_r))\right)$.

φ_l and φ_r are two functions that reorganize the inputs to match the types of c_1 and c_2. They are defined as $\varphi_l\ (a, (b, c)) := (a, b)$ and $\varphi_r\ (a, (b, c)) := ((a, b), c)$.

Example 4.2. When applying to $\mathsf{LP}(mss, mss)$, f_1 and f_2 can be $mps\triangle mts$ and sum respectively. At this time, the structures of the inputs of c, c_1 and c_2 are listed below.

- (c) $\left(\left(mss, ((mps, mts), sum)\right), \left(mss, ((mps, mts), sum)\right)\right)$
- (c_1) $\left((mss, (mps, mts)), (mss, (mps, mts))\right)$
- (c_2) $\left(\left((mss, (mps, mts)), sum\right), \left((mss, (mps, mts)), sum\right)\right)$

It is easy to verify that the structure of the inputs of c, $c_1 \circ (\varphi_l \times \varphi_l)$ and $c_2 \circ (\varphi_r \times \varphi_r)$ are equal.

Theorem 4.3 (Correctness of Decomposition)**.** *Any result found by rule decomposition is a valid solution for* $\mathsf{LP}(p, h)$.

Due to the space limit, we omit the proofs to theorems, which can be found in Appendix C.

Note that *decomposition* can still be applied to the second subtask $\mathsf{LP}(f_1, h \triangle f_1)$. Therefore, a lifting problem can be completely converted into partial lifting problems by recursively applying rule *decomposition*.

For efficiency, *AutoLifter* considers only the first solution to $\mathsf{PLP}(p, h)$ while applying *decomposition*. Such an implementation is incomplete. It is possible that the first synthesized lifting scheme is invalid, and a correct solution can never be found. However, as we shall show in Section 7, on

all tasks in our evaluation this greedy approach works well without backtracking. Besides, we also provide an implementation of *decomposition* that ensures completeness, which can be found in Appendix B.1.

Decoupling. Given partial lifting problem $\mathsf{PLP}(p, h)$, the procedure of *decoupling* is listed below.

- Solve subtask $\mathsf{S}_f(p, h)$. Let f be the result.
- Solve subtask $\mathsf{S}_c(p, h, f)$. Let c be the result.
- Take (f, c) as the synthesis result.

The following theorem shows the correctness of this rule.

Theorem 4.4 (Correctness of Decoupling). *Any (f, c) found by rule decoupling is a valid solution for $\mathsf{PLP}(p, h)$.*

When the program space for c (i.e., grammar G_c) is expressive enough to represent any possible combinator, rule *decoupling* is also complete. Such a conclusion is proven in Appendix C as Lemma C.11. But in practice, the expressiveness of G_c is usually limited. At this time, $\mathsf{S}_f(p, h)$ is only a weak specification over f and *decoupling* becomes incomplete. It is possible that there is no valid combinator corresponding to f synthesized from $\mathsf{S}_f(p, h)$.

There are two important properties making *decoupling* effective in practice. At first, all programs related to $\mathsf{S}_f(p, h)$, including p, h and candidate programs in G_f, map a list to a constant number of scalar values. Therefore, their input spaces are far larger than the output spaces, i.e., they are *compressing*. According to the definition, $\mathsf{S}_f(p, h)$ requires f to output differently on inputs satisfying some condition with respect to p and h. The small output space makes an incorrect program hardly to satisfy $\mathsf{S}_f(p, h)$, since the smaller the output space is, the more likely for a program to output the same on two different inputs.

Second, in *AutoLifter*, the generalizability of the synthesizer for $\mathsf{S}_f(p, h)$ is guaranteed by *Occam solvers* [Ji et al. 2021], and thus *AutoLifter* can find the user-wanted lifting scheme f from $\mathsf{S}_f(p, h)$ in practice. More details on *Occam solvers* can be found in Section 4.4.

We formalize the relationship between these two properties and the effectiveness of *decoupling* in Appendix C as Theorem C.3. This theorem demonstrates that when both two above properties hold and the semantics are modeled as random, the probability for *AutoLifter* to synthesize an unwanted lifting scheme from S_f is negligible.

The practical performance of *AutoLifter* matches the theoretical analysis. For all tasks in our evaluation, there is always a valid combinator corresponding to the first solution found by *AutoLifter* for S_f. We also provide an implementation of *decoupling* that ensure completeness in Appendix B.1.

4.3 Inductive Part

AutoLifter solves subtasks of type S_f and S_c via inductive synthesizers. The synthesis algorithms for both types are comprised of a synthesizer and a verifier and are under the framework of *counter-example guided inductive synthesis* [Solar-Lezama et al. 2006]. In this subsection, we introduce the synthesizers, and the verifiers will be left to Section 4.4.

We take *PolyGen* [Ji et al. 2021], a state-of-the-art synthesizer based on input-output examples, as the synthesizer for S_c. Because *PolyGen* is already effective enough for most known tasks, we do not modify its synthesis algorithm.

We build the synthesizer for S_f based on *observational equivalence* (denoted as $\mathcal{O}_e$) introduced in Section 3. We use order $<_s$ to represent the enumeration order of $\mathcal{O}_e$, where $f_1 <_s f_2$ represents that f_1 is visited before f_2 by $\mathcal{O}_e$.

Because $\mathcal{O}_e$ is not effective enough for many known tasks, we improve it via a special treatment for operator $\triangle$, namely *observational covering* (denoted as $\mathcal{O}_c$). We regard the lifting scheme as an ordered list of lifting functions, which are also programs in G_f. To distinguish, we call the list representation of a lifting scheme as a *composed program*, which is a list $\overline{f} = [f_1, \ldots, f_k]$ of lifting programs $f_1 \ldots f_k \in G_f$ satisfying $f_k <_s \cdots <_s f_1$. A composed program $[f_1, \ldots, f_k]$ can be converted into a lifting scheme by concatenating $f_1, \ldots, f_k$ via the product operator $\triangle$, resulting in $f_1 \triangle \ldots \triangle f_k$.

Similar with $\mathcal{O}_e$, $\mathcal{O}_c$ firstly sets up a goal for finding the smallest composed program and then uses pruning strategies to skip those programs that are impossible to be optimal. In $\mathcal{O}_m$, the smallest is defined via a partial order over composed programs, namely $\prec_c$, where $\overline{f} = [f_1, \ldots, f_k] \prec_c \overline{f'} = [f'_1, \ldots, f'_{k'}]$ if k is no larger than k' and list $\overline{f}$ is lexicographically smaller than list $\overline{f'}$ according to $<_s$. The target of $\mathcal{O}_c$ is to find a minimal composed program in the sense of $\prec_c$.

$\mathcal{O}_c$ is based on the concept of *observationally covered* (abbreviated as *covered*) programs (Definition 4.5). Because a minimum of $\prec_c$ must be uncovered, programs that are covered can be skipped. Lemma 4.6 describes a pruning strategy based on this point.

Definition 4.5. $\overline{f}$ is said to be *observationally covered* on example set E if $\exists \overline{f'} \prec_c \overline{f}, E|_{\overline{f}} \subseteq E|_{\overline{f'}}$, where $E|_{\overline{f}}$ represents the set of examples satisfied by some program in $\overline{f}$.

Lemma 4.6. *Composed program $\overline{f}$ is uncovered on $E \Rightarrow \forall \overline{f'} \subseteq \overline{f}$, $\overline{f'}$ is uncovered on E, where $\overline{f_1} \subseteq \overline{f_2}$ represents that all lifting functions in $\overline{f_1}$ are in $\overline{f_2}$ as well.*

Algorithm 1 shows the pseudo-code of $\mathcal{O}_c$. We maintain a working list (*workingList*) that queues composed programs to be enumerated and a list (*minList*) containing existing uncovered programs. In each turn, we either obtain a new composed program from $\mathcal{O}_e$ (line 6) or obtain a composed program from the working list (lines 7-8). If this program is uncovered (Line 18) and is not yet a solution (Line 19), it will be used to construct new programs (Lines 11-13, 20).

There are two noticeable points in Algorithm 1. First, because verifying uncovered programs according to Definition 4.5 is time-consuming (Line 4), $\mathcal{O}_c$ firstly use Lemma 4.6, a

Algorithm 1: The pseudo code of synthesizer $\mathcal{O}_c$.

Input: A set E of examples and parameter n_c.
Output: Lifting scheme f^* satisfying all examples.

1 $\forall size \geq 1$, $minList[size] \leftarrow []$, $workingList[size] \leftarrow []$;
2 **Function** CheckUnCovered($size$, $program$):
3 **if** $\exists \overline{f'} \subset \overline{f}, \overline{f'} \notin minList$ **then return** *false*;
4 **return** $\forall k \in [1, size], \forall \overline{f} \in minList[k], E|_{\overline{f}} \not\subseteq E|_{program}$;
5 **Function** NextComposedProgram($size$):
6 **if** $size = 1$ **then return** $[\mathcal{O}_e$.Next()$]$;
7 **if** $workingList[size]$.Empty() **then return** *null*;
8 **return** $workingList[size]$.PopFront();
9 **Function** InsertNewPrograms($size$, $prog = [f_1, \ldots, f_{size}]$):
10 $minList[size]$.PushBack($prog$);
11 **for** *each* $[f] \in minimalList[1]$ *satisfying* $f <_s f_{size}$ **do**
12 $\overline{f} \leftarrow [f_1, \ldots, f_{size}, f]$;
13 $workingList[size + 1]$.PushBack($\overline{f}$);
14 **end**
15 **for** $turn \leftarrow 1 \ldots + \infty$ **do**
16 $s \leftarrow (turn - 1) \bmod n_c + 1$;
17 $\overline{f} \leftarrow$ NextComposedProgram(s);
18 **if** $\overline{f} = null \vee \neg$CheckUnCovered($s, \overline{f}$) **then continue**;
19 **if** $E|_{\overline{f}} = E$ **then return** $f_1 \triangle \ldots \triangle f_s$ $(\overline{f} = [f_1, \ldots, f_s])$;
20 InsertNewPrograms($s, \overline{f}$);
21 **end**

necessary condition for uncovered programs, to preclude programs (Line 3). Second, parameter n_c is used to control the number of lifting functions. $\mathcal{O}_c$ ensures that each number in $[1, n_c]$ is considered with the same frequency (Line 16). Such a limit ensures $\mathcal{O}_c$ to be an Occam solver, as will be discussed in Section 4.4.

Theorem 4.7 (Properties of Observational Covering). *Given task $S_f(p, h)$ and a set E of examples, let S be the set of all valid composed programs. When S is non-empty, $\mathcal{O}_c$ always **terminates**. Besides, the program $\overline{f^*}$ synthesized by $\mathcal{O}_c$ always satisfies (1) **validity**: $\overline{f^*} \in S$, (2) **minimality**, $\forall \overline{f} \in S, \neg\left(\overline{f} <_c \overline{f^*}\right)$.*

4.4 Verification

In this section, we introduce our verifier. We start with an introduction to *Occam solver* [Ji et al. 2021].

Given a set of synthesis tasks $\mathbb{T}$ and constants $\alpha \geq 1, 0 \leq \beta < 1$, solver $\mathcal{S}$ is an (α, β)-Occam solver on $\mathbb{T}$ if for any task $T \in \mathbb{T}$, any set E of examples and any correct program p^*, the size of the program synthesized by $\mathcal{S}$ is always no larger than $c(\text{size}(p^*))^\alpha |E|^\beta$, where c is a large enough constant and size(p) is the length of the binary representation of p.

PolyGen is an Occam solver and the following theorem shows that $\mathcal{O}_c$ is also an Occam solver. Therefore, both solvers for S_f and S_c in *AutoLifter* are Occam solvers.

Theorem 4.8. *$\mathcal{O}_c$ (Algorithm 1) is a $(1, 0)$-Occam solver.*

The correctness of an Occam solver is guaranteed in a probabilistic way [Blumer et al. 1987]. When the number of examples is polynomial to the size of the smallest valid program, the probability for the generalization error of the synthesis result to exceed a threshold is bounded.

Therefore, to obtain a probabilistic guarantee on the correctness, we only need to ensure the number of examples is enough. The verifier of *AutoLifter* achieves it by iterating with the number of examples n_s and a threshold t. Given a synthesis task T, a solver $\mathcal{S}$, and a generator that independently generates examples from a fixed distribution D, the verifier executes in the following way:

1. Set t to 1, and set n_s to a pre-defined parameter n_0.
2. Invoke $\mathcal{S}$ using n_s samples. Let p be the synthesis result.
3. Return p as the result if size(p) $\leq t$.
4. Double n_s and t, and go back to step 2.

The following theorem shows that this iterative algorithm provides a probabilistic guarantee on the correctness and it must terminate when $\mathcal{S}$ is an Occam solver.

Theorem 4.9 (Probabilistic Guarantee of the Verifier). *For any synthesis task T, any solver $\mathcal{S}$ available for task T, and any distribution D for examples, if the iterative algorithm terminates, the synthesized program p always satisfies the following formula.*

$$\forall \epsilon \in (2 \ln 2/n_0, 1), \Pr\left[err_D(p) \geq \epsilon\right] \leq 4 \exp(-\epsilon n_0)$$

where $err_D(p)$ represents the probability for p to violate an example drawn from distribution D.

Moreover, the iterative algorithm always terminates when $\mathcal{S}$ is an Occam solver.

5 Lifting Problem

In this section, we introduce the concept of *lifting problem* and show how *AutoLifter* generalizes to other tactics in brief.

5.1 Motivation

Let us see the application conditions of several other tactics. Due to the space limit, we introduce these tactics in brief. More details can be found in Appendix D.

- $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}, \mathcal{A}_{l,3}$ represent three greedy algorithms for *longest segment problem* (LSP) [Zantema 1992], which is described by a predicate b on lists. Given list l, LSP(b) queries the length of the longest segment in l satisfying b. Application conditions for $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}, \mathcal{A}_{l,3}$ are Formula 2, Formula 2 and 3, and Formula 4 respectively.

$$(b\ (l +\!\!+ [a]), f\ (l +\!\!+ [a])) = c\ ((b\ l, f\ l), a) \quad (2)$$

$$(b\ (tail\ l), f\ (tail\ l)) = c'\ ((b\ l, f\ l), a) \quad (3)$$

$$(p\ (l_1 +\!\!+ [a] +\!\!+ l_2), f\ (l_1 +\!\!+ [a] +\!\!+ l_2)) = c\ ((p\ l_1, f\ l_1), a, (p\ l_2, f\ l_2)) \quad (4)$$

where p represents a program for task LSP(b).

- $\mathcal{A}_r$ is *lazy propagation*, a classical algorithm, for *range update and range query* (RANGE) [Lau and Ritossa 2021]. Given list x, query function h, update function u, and a list of operations o_i, the task is to process operations in order.
  - Update $o = (\mathsf{U}, a, l, r)$: set x_i to $u\ (a, x_i)$ for each $i \in [l, r]$.
  - Query $o = (\mathsf{Q}, l, r)$: calculate and output $h\ [x_l, \ldots, x_r]$.

  The following shows the application condition of $\mathcal{A}_r$.

$$(h\ (map\ u_a\ l), f\ (map\ u_a\ l)) = c_1\ ((h\ l, f\ l), a)$$
$$(h\ (l_1 +\!\!+ l_2), f\ (l_1 +\!\!+ l_2)) = c_2\ ((h\ l_1, f\ l_1), (h\ l_2, f\ l_2))$$

  where u_a is the abbreviation of $\lambda w.u\ (a, w)$.
- $\mathcal{A}_m$ represents an algorithm using dynamic programming for *maximum weightsum problem* (MMP) [Sasano et al. 2000], which is described by a predicate b. Given a weighted list, the task of MMP(b) is to mark a sublist satisfying b and maximize the weight-sum of marked elements. The application condition of $\mathcal{A}_m$ has the same form as Formula 2.

The application conditions of all the above tactics and the tactic for divide-and-conquer are similar in the form.

- On the left, a known program p (or b) and an unknown lifting scheme f is applied to the input.
- On the right, p (or b) and f are applied to lists in the input, and the results are merged via an unknown combiantor c.

The generalized lifting problem abstracts this from and thus includes the synthesis tasks of these mentioned tactics.

5.2 Problem Definition

We use functors to define the lifting problem. A functor (denoted by F) maps types to types, functions to functions, and keeps both identity and composition.

$$\mathsf{F}id_A = id_{\mathsf{F}A} \quad \mathsf{F}(f \circ g) = \mathsf{F}f \circ \mathsf{F}g$$

In this paper, we only consider functors constructed by identity functor I, constant functors $!A$ for any type A, and bifunctor $\times$. Their definitions are shown below.

$$(!A)B = A \quad (!A)f = id_A \quad \mathsf{I}A = A \quad \mathsf{I}f = f$$
$$(\mathsf{F}_1 \times \mathsf{F}_2)A = (\mathsf{F}_1 A) \times (\mathsf{F}_2 A) \quad (\mathsf{F}_1 \times \mathsf{F}_2)f = (\mathsf{F}_1 f) \times (\mathsf{F}_2 f)$$

We further define a specific class of functions, *constructors*, which capture the different ways of constructing the input to p and f. Given a type A, a constructor m for A is a function with an attached functor F_m such that the signature of m is $F_m A \to A$.

With the above notations, we define the lifting problem.

Definition 5.1 (Lifting Problem)**.** Let p be a program from A to B and M be a set of constructors $\{m_1, \ldots, m_n\}$. Lifting problem LP(M, p) is to find a lifting scheme f and n combinators $c_1, \ldots, c_n$ satisfying the formula.

$$\forall m_i \in M, (p \triangle f) \circ m_i = c_i \circ \mathsf{F}_{m_i}(p \triangle f)$$

Example 5.2. The synthesis tasks corresponding to divide-and-conquer and tactics discussed in Section 5.1 can be regarded as the following lifting problems.

- (Divide-and-Conquer) LP$(\{\lambda(l_1, l_2).l_1 +\!\!+ l_2\}, p)$.
- $(\mathcal{A}_{l,1}, \mathcal{A}_m)$ LP$(\{\lambda(l, a).l +\!\!+ [a]\}, b)$.
- $(\mathcal{A}_{l,2})$ LP$(\{\lambda(l, a).l +\!\!+ [a], \lambda(l, a), (tail\ l)\}, b)$.
- $(\mathcal{A}_{l,3})$ LP$(\{\lambda(l_1, a, l_2).l_1 +\!\!+ [1] +\!\!+ l_2\}, p)$.
- $(\mathcal{A}_r)$ LP$(\{\lambda(l, a).map\ u_a\ l, \lambda(l_1, l_2).l_1 +\!\!+ l_2\}, h)$.

5.3 Generalization of *AutoLifter*

Now, we show how to generalize *AutoLifter* to lifting problems in brief. More details can be found in Appendix A.

Subtasks We first redefine the subtasks in Section 4.1.

- Lifting problem LP(M, p, h) for synthesizing f and c_i for each constructor M.

$$\forall m_i \in M, (p \triangle f) \circ m_i = c_i \circ \mathsf{F}_{m_i}(p \triangle f)$$

- Partial lifting problem PLP(M, p, h) for synthesizing f and c_i for each constructor in M.

$$\forall m_i \in M, p \circ m_i = c_i \circ \mathsf{F}_{m_i}(p \triangle f)$$

  and its special case SPLP(m, p, h) when $M = \{m\}$.
- Subtask $\mathsf{S}_f(m, p, h)$ for f and subtask $\mathsf{S}_c(m, p, h, f)$ for c.

$$\left(\mathsf{F}_m h\ \bar{i} = \mathsf{F}_m h\ \bar{i'} \wedge p\ (m\ \bar{i}) \neq p\ (m\ \bar{i'})\right) \to \mathsf{F}_m f\ \bar{i} \neq \mathsf{F}_m f\ \bar{i'}$$

$$p \circ m = c \circ \mathsf{F}_m(p \triangle f)$$

Deductive Part. In Section 4, *decomposition* splits the product on the left of the specification, and *decoupling* decouples the functional composition on the right. The generalized subtasks retain these two substructures. Therefore, rule *decomposition* and *decoupling* can be naturally generalized to convert LP(M, p, h) into partial lifting problems, and convert SPLP(m, p, h) into S_f and S_c, respectively.

Similar to the divide-and-conquer version, the effectiveness of *decoupling* in *AutoLifter* is guaranteed when p and available lifting schemes all map their input space to a far smaller output space, e.g., from a recursive data structure to a tuple of scalar values.

There remains a gap between PLP and SPLP. A PLP task with n constructors can be split into n SPLP tasks, each dealing with one constructor. Their results can be merged by taking the joint of all used lifting functions and adjusting the inputs of combinators according to the type.

Inductive Part. *PolyGen* can be applied to S_c because input-output examples for c is available, and *observational covering* can be applied to S_f as it is still common for f to include multiple lifting functions under the generated setting.

Verification. The iterative verification requires (1) the existence of a generator for examples, (2) the solvers to be Occam solvers. Both of them hold under the generalized setting.

6 Implementation

Our implementation can be found in the supplementary material. The current implementation requires that the original program and available lifting schemes all map a list to a tuple of scalar values. *AutoLifter* can be applied to other data structures if corresponding operators and grammars are provided.

Grammars. We take the grammar used by *DeepCoder* [Balog et al. 2017] as G_f. This grammar contains 17 list operating functions, including commonly used higher-order functions such as *map* and *filter*, and operators that perform branching and looping internally, such as *sort* and *count*.

We take the grammar for conditional integer arithmetic in SyGuS-Comp [Alur et al. 2019] as G_c. This grammar contains basic arithmetic operators such as $+, -, \times$, *div*, Boolean operators, and branch operator *if-then-else*. G_c can express complex programs via nested *if-then-else* operators.

The complete grammars can be found in Appendix B.2.

Parameters for Verifiers. Distribution D defines how the examples are sampled. To avoid arithmetic overflow while using $\times$, we let D focus on short lists and small integers:

- For type `List`, D draws an integer from $[0, 10]$ as its length, and recursively samples the contents.
- For type `Int`, D draws an integer from $[-5, 5]$.
- For type `Bool`, D draws a value from {true, false}.

Parameter n_0 determines the initial number of examples. Guided by Theorem 4.9, we set n_0 to 10^4. At this time, the probability for the generation error of the synthesized program to be more than 0.001 is at most 1.82×10^{-4} .

Other Parameters. While implementing $\mathcal{O}_c$, we set n_c to 4 because the product of four lifting functions is 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* synthesize divide-and-conquer programs?
- **RQ2**: How does *AutoLifter* generalizes to other tactics?

7.1 Baseline Solvers

First, we compare *AutoLifter* with a SOTA synthesizer for divide-and-conquer, *Parsynt* [Farzan and Nicolet 2017, 2021]. *Parsynt* is a white-box solver, requiring the original program to be single-pass. It uses pre-defined rules to transform the loop body, extract the lifting functions directly, and then synthesize the combinator via inductive synthesizers.

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 2021], and consider both of them in evaluation.

Second, we compare *AutoLifter* with two weakened solvers.

- *Enum* is an enumerative solver. Given task $\mathsf{LP}(M, p)$, *Enum* enumerates candidate solution $(f, \{c_i\})$ in the increasing order of the total size, until a correct solution is found.
- *DEnum* is weakened from *AutoLifter* by replacing *observational covering* with *observational equivalence*.

First, comparing with *Enum* and *DEnum* forms ablation studies on the deductive system and *observational covering* respectively. Second, as shown in Section 8, several existing synthesizers degenerates to *DEnum* on lifting problems.

7.2 Dataset

Our evaluation is conducted on two datasets $\mathcal{D}_D$ and $\mathcal{D}_L$.

Dataset $\mathcal{D}_D$. The first dataset $\mathcal{D}_D$ is based on the datasets used by previous work on program calculation and algorithm synthesis [Bird 1989; Farzan and Nicolet 2017, 2021][2]. $\mathcal{D}_D$ includes all tasks (36 in total) that target to synthesize divide-and-conquer-style parallel program from these benchmarks. As a result, $\mathcal{D}_D$ contains all tasks used by Bird [1989]; Farzan and Nicolet [2017] and 12 out of 22 tasks used by Farzan and Nicolet [2021]. The other 10 tasks involve a class of divide-and-conquer different from the tactic proposed by Cole [1995], and thus are out of the scope of *AutoLifter*.

Dataset $\mathcal{D}_L$. The second dataset $\mathcal{D}_L$ consists of 21 tasks that are related to tactics $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}, \mathcal{A}_{l,3}$ for problem LSP and tactic $\mathcal{A}_r$ for problem RANGE. These tactics and problems have been introduced in Section 5.1.

For LSP, $\mathcal{D}_L$ contains all samples used by Zantema [1992], including 3, 1, 4 tasks for $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}, \mathcal{A}_{l,3}$ respectively.

For RANGE, because no previous work on *lazy propagation* provides a dataset, we searched *Codeforces*, an online platform for competitive programming, using "segment tree" and "lazy propagation", and obtained 13 tasks in $\mathcal{D}_L$.

There is another tactic $\mathcal{A}_m$ mentioned in Section 5.1. We do not consider it in $\mathcal{D}_L$ because the efficiency of its result is related to the range of the lifting functions and thus is not supported by our current implementation. *AutoLifter* can be extended to this tactic by skipping those results where the range is large in $\mathcal{O}_c$.

Guarantees. For all 5 tactics involved in $\mathcal{D}_D$ and $\mathcal{D}_L$, the efficiency of the resulting program is guaranteed if the combinator is constant-time. As discussed in Section 3, *AutoLifter* guarantees this point by two assumptions on the grammars.

7.3 RQ1: Comparison on Divide-and-Conquer

Procedure. We compare *AutoLifter* with baseline solvers on tasks in $\mathcal{D}_D$ with a time limit of 300 seconds and a memory limit of 8 GB. There are three noticeable points in the setting.

[2]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 program into a single-pass function with *fold*. These bugs were confirmed by the original authors, and we fixed them in our evaluation. This also demonstrates that writing a program as a single-pass function is difficult and error-prone.

- The default grammars discussed in Section 6 are not expressive enough for 8 tasks in $\mathcal{D}_D$, where operators such as regex matching on an integer list are required. Therefore, we set up an **enhanced setting** where missing operators are manually provided to the grammars. Details on these operators can be found in Appendix E.1.
- To invoke *Parsynt*, we provide single-pass implementations for tasks in $\mathcal{D}_D$[3]. At this time, some lifting functions have already been involved, as discussed in Section 3.1.
- We failed in installing *Parsynt17* because of some issues on the dependencies. The authors of *Parsynt17* confirmed but have not solved this problem. So we compare *AutoLifter* with *Parsynt17* on its original dataset $\mathcal{D}_D^-$ using the evaluation results reported by Farzan and Nicolet [2017].

Results. The results of this experiment are summarized as the upper part of Table 1. We manually verify all synthesis results and find that *AutoLifter* always returns a **completely correct** solution on those solved tasks.

On efficiency, *AutoLifter* solves no fewer tasks than all baselines under all settings and usually solves much more. Note that though about 60% and 40% lifting functions are directly provided to *Parsynt17* and *Parsynt21* respectively, *AutoLifter* still solves a competitive number of tasks and achieves a much faster speed on those jointly solved tasks.

One interesting result is that *AutoLifter* uses fewer lifting functions than both versions of *Parsynt*. One reason is that the syntax may mislead *Parsynt* to some unnecessarily complex solutions. We take task *line_sight* (abbreviated as *ls*) as an example, which checks whether the last element is the maximum in a list. $(ls \triangle max)\ l = fold\ \oplus\ (false, -\infty)\ l$ is a single-pass implementation for *ls* with lifting function *max*, where $(ls_1, max_1) \oplus a := (a \geq max_1, \max(a, max_1))$. Because there is a comparison between $\max_1$ and a, the last visited value, *Parsynt* takes *last* l as a lifting function. However, such a lifting function is unnecessary, because $ls\ (l_1 +\!\!+ l_2) = (ls\ l_2) \wedge (max\ l_1 \leq max\ l_2)$. *AutoLifter* can find this solution as it synthesizes from the semantics.

Under the enhanced setting, the performance of *AutoLifter* is improved. Such a result shows that *AutoLifter* can be further improved if missing operators can be automatically inferred. To achieve this, one possible way is to extract useful operators from the original program. This will be future work. The only failed task is *longest_odd_(0+1)* constructed by Farzan and Nicolet [2021], where *AutoLifter* successfully finds a correct lifting scheme but *PolyGen* fails.

7.4 RQ2: Comparison on Other Tactics

Procedure. We compare *AutoLifter* with *Enum*, *DEnum* on tasks in $\mathcal{D}_L$ with a time limit of 300 second and a memory limit of 8 GB. Similar with Section 7.3, we manually provide missing operators for one task under the enhanced setting.

[3]For tasks taken from *Parsynt*, we use the program in its original evaluation

Result. The results of this experiment are summarized as the lower part of Table 1, which demonstrates the effectiveness of *AutoLifter*. Under the enhanced setting, *AutoLifter* solves all tasks with an average time cost of 7.52 seconds.

7.5 Case Study

We make a case study on two tasks in our dataset. Due to the space limit, we only list the results of the case study here. More details can be found in Appendix E.2.

- The first shows the advantage of black-box synthesis. This task is for the maximum but requires lifting functions that calculate the minimum. *Parsynt* fails on this task because this solution is counter-intuitive on syntax and is out of the scope of the transformation rules in *Parsynt*. In contrast, *AutoLifter* successfully solves it by directly synthesizing from the semantics.
- The second shows that *AutoLifter* is able to solve tasks difficult for human programmers. This task was used by the 2020-2021 Winter Petrozavodsk Camp, which is a worldwide training camp representing the highest level of competitive programming. Only 26 out of 243 teams successfully solved this task in 5 hours, while *AutoLifter* solves it using only 14.33 seconds.

8 Related Work

Algorithm Synthesis. Many existing approaches have been proposed to automatically synthesize divide-and-conquer-style parallel programs [Ahmad and Cheung 2018; Farzan and Nicolet 2017; Fedyukovich et al. 2017; Morita et al. 2007; Radoi et al. 2014; Raychev et al. 2015; Smith and Albarghouthi 2016]. Some approaches [Farzan and Nicolet 2017; Fedyukovich et al. 2017; Morita et al. 2007; Raychev et al. 2015] support to find lifting functions. However, all of them require the original program to be single-pass. To our knowledge, *AutoLifter* is the first that synthesizes the lifting functions without requiring a single-pass implementation.

There are two solvers for divide-and-conquer that go beyond the algorithmic tactic used in this paper. Farzan and Nicolet [2019] and Farzan and Nicolet [2021] support programs with nested loops, and the latter also support a more general class of divide operators other than dividing at the middle. They are based on single-pass programs and thus their contributions are orthogonal to ours.

There are also approaches for other algorithms. Lin et al. [2019] target at dynamic programming and Acar et al. [2005] incrementalizes an existing program. Both of them also require the syntax of the original program. *AutoLifter* are not designed for these algorithms as the application conditions in their tactics are not in the form of lifting problems.

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

Table 1. The results of the evaluation.

Solver	Dataset	#Tasks	#Solved		Average Time Cost (s)[1]		Average #Supplementaries[1,3]	
			Baseline	*AutoLifter*	Baseline	*AutoLifter*	Baseline	*AutoLifter*
Exp1 (Section 7.3)[2]								
AutoLifter	$\mathcal{D}_D$	36	28 (35)		7.11 (8.90)		2.04 (2.31)	
Enum			5 (6)	**28 (35)**	7.41 (9.89)	**0.05 (0.40)**	0.20 (0.33)	0.20 (0.33)
DEnum			14 (15)		9.64 (9.30)	**3.74 (3.63)**	1.14 (1.13)	1.14 (1.13)
Parsynt17	$\mathcal{D}_D^-$	20	19	19 (**20**)	15.59 (19.22)	**3.85 (3.76)**	0.94+0.61 (0.89+0.63)	**1.5 (1.47)**
Parsynt21	$\mathcal{D}_D$	36	24	**28 (35)**	5.62 (6.74)	**1.19 (5.46)**	1.21+1.58 (1.25+1.79)	**1.79 (2.21)**
Exp2 (Section 7.4)[2]								
AutoLifter	$\mathcal{D}_L$	21	20 (21)		7.80 (7.52)		2.25 (2.29)	
Enum			4 (4)	**20 (21)**	67.92 (67.92)	**0.20 (0.20)**	1.00 (1.00)	1.00 (1.00)
DEnum			15 (16)		11.64 (12.65)	**7.00 (6.68)**	1.53 (1.63)	1.53 (1.63)

[1] The average includes only the results on the tasks solved by both solvers.
[2] For results in the form of a (b), a and b represents results under the default setting and the enhanced setting respectively.
[3] The number of lifting functions used by *ParSynt* is listed in the form of $c + d$, where c and d represents the number of manually provided lifting functions and the number of synthesized lifting functions respectively.

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 algorithm synthesis, these approaches can achieve more refined guarantees 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 the algorithms our approach does.

Program Synthesis. Program synthesis is an active field and many synthesizers have been proposed. Here we discuss the most-related approaches. *AutoLifter* is related to *DryadSynth* [Huang et al. 2020], which also combines deductive methods and inductive methods. However, the deductive rules in *DryadSynth* are based on Boolean/arithmetic operators, and thus are useless for lifting problems, where these operators are not explicitly used in the specification. Moreover, because *DraySynth* uses *observational equivalence* as the inductive solver, it will work the same as our baseline *DEnum* if we replace its deductive system with *AutoLifter*'s.

AutoLifter is also related to *Relish* [Wang et al. 2018], which targets to specifications with multiple unknown functions. *Relish* builds *finite tree-automates (FTA)* for variables and synthesizes by merging them together. However, as the time cost of constructing an FTA is similar to *observational equivalence*, *Relish* cannot be much faster than our baseline *DCEnum*.

There are also solvers for synthesizing list-operating programs, including *DeepCoder* [Balog et al. 2017], *Myth* [Osera and Zdancewic 2015], λ^2 [Feser et al. 2015] and *Refazer* [Rolim et al. 2017]. All of these solvers are based on input-output examples, which are unavailable in task S_f.

9 Conclusion

In this paper, motivated by the difficulty of implementing single-pass programs, we study the problem of black-box algorithm synthesis for divide-and-conquer. Inspired by the previous work on program calculus, we use algorithmic tactics to convert the task of algorithm synthesis to the task of solving the corresponding application condition.

To efficiently solve the application condition of divide-and-conquer, we propose a novel synthesizer *AutoLifter*. *AutoLifter* first uses two deductive rules, *decomposition* and *decoupling*, to reduce the scale of the result, and then uses inductive synthesizers, *PolyGen* and *observational covering*, to solve the subtasks generated by the deductive rules. *AutoLifter* uses grammars and an iterative verifier to guarantee the efficiency and the correctness of the synthesis result.

We also generalize *AutoLifter* to other algorithms where the application condition follows the same structure with divide-and-conquer. We define the lifting problem to capture the commonality of these algorithms and generalize *AutoLifter* to the general lifting problems. In this way, *AutoLifter* can be applied to all those algorithms where the application condition is an instance of the lifting problem.

Our evaluation demonstrates the effectiveness of *AutoLifter* on lifting problems and shows that though *AutoLifter* does not access the syntax of the original program, it still achieves competitive performance compared with state-of-the-art white-box approaches for divide-and-conquer.

References

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

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

Jon Louis Bentley. 1977. Solutions to KleeâĂŹs rectangle problems. *Unpublished manuscript* (1977), 282–300.

Richard Bird. 1989. Lecture notes in Theory of Lists.

Richard S. Bird and Oege de Moor. 1997. *Algebra of programming*. Prentice Hall.

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 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. 2019. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 610–624. https://doi.org/10.1145/3314221.3314612

Azadeh Farzan and Victor Nicolet. 2021. 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

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

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Occam Learning Meets Synthesis Through Unification. *CoRR* abs/2105.14467 (2021). arXiv:2105.14467 https://arxiv.org/abs/2105.14467

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

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

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

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 107–126. https://doi.org/10.1145/2814270.2814310

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

Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. 2000. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. 137–149. https://doi.org/10.1145/351240.351254

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

A Appendix: Generalized Approach

In this section, we elaborate the detail on generalizing *AutoLifter* to the lifting problems, defined in Section 5.2.

Similar to the main text, the proofs of the theorems in this section can be found in Appendix C.

A.1 Subtasks

The generalized subtasks has been introduced in Section 5.3. They are defined in the following way.

- Lifting problem $\mathsf{LP}(M = \{m_1, \ldots, m_n\}, p, h)$.

$$\forall m_i \in M, (p \triangle f) \circ m_i = c_i \circ \mathsf{F}_{m_i}(p \triangle f)$$

- Partial lifting problem $\mathsf{PLP}(M = \{m_1, \ldots, m_n\}, p, h)$,

$$\forall m_i \in M, p \circ m_i = c_i \circ \mathsf{F}_{m_i}(p \triangle f)$$

  and its special case $\mathsf{SPLP}(m, p, h)$ when $M = \{m\}$.
- Subtask $\mathsf{S}_f(m, p, h)$ and $\mathsf{S}_c(m, p, h, f)$.

$$\left(\mathsf{F}_m h\ \bar{i} = \mathsf{F}_m h\ \bar{i'} \wedge p\ (m\ \bar{i}) \neq p\ (m\ \bar{i'})\right) \rightarrow \mathsf{F}_m f\ \bar{i} \neq \mathsf{F}_m f\ \bar{i'}$$

$$p \circ m = c \circ \mathsf{F}_m(p \triangle f)$$

A.2 Deductive Part

Decomposition. This rule is the generalization of rule *decomposition* discussed in Section 4. Given lifting problem $\mathsf{LP}(M = \{m_1, \ldots, m_n\}, p, h)$, its procedure is listed below.

1. Solve subtask $\mathsf{PLP}(M, p, h)$ and get $(f, c_1, \ldots, c_n)$.
2. Return $(f, c_1 \triangle null, \ldots, c_n \triangle null)$ when $f = null$.
3. Solve subtask $\mathsf{LP}(M, f, h \triangle f)$ and get $(f', c_1', \ldots, c_n')$.
4. Return $(f \triangle f', c_1^*, \ldots, c_n^*)$. c_i^* is defined as the following.

$$c_i^* \coloneqq \left(c_i \circ \mathsf{F}_{m_i}\varphi_l\right) \triangle \left(c_i' \circ \mathsf{F}_{m_i}\varphi_r\right)$$

The definitions of φ_l and φ_r remains unchanged, where $\varphi_l\ (a, (b, c)) \coloneqq (a, b)$ and $\varphi_r\ (a, (b, c)) \coloneqq ((a, b), c)$.

Theorem A.1 (Correctness of Decomposition). *Any result found by decomposition is a valid solution for* $\mathsf{LP}(M, p, h)$.

Decomposition*. This is a supplementary rule for converting a general PLP task into the special case SPLP. Given a partial lifting problem $\mathsf{SPLP}(M = \{m_1, \ldots, m_n\}, p, h)$, the procedure of *decomposition** is listed below.

1. $\forall i \in [1, n]$, solve subtask $\mathsf{SPLP}(m_i, p, h)$ and get (f_i, c_i).
2. Return $(f_1 \triangle \ldots \triangle f_n, c_1 \circ \mathsf{F}_{m_1}\varphi_1, \ldots, c_n \circ \mathsf{F}_{m_n}\varphi_n)$.

Function φ_i reorganizes the inputs for $c_1, \ldots, c_n$, which is defined as $\varphi_i\ (a, (b_1, \ldots, b_n)) \coloneqq (a, b_i)$.

Theorem A.2 (Correctness of Decomposition*). *Any result found by decomposition* is a valid solution for* $\mathsf{PLP}(M, p, h)$.

Decoupling. Given partial lifting problem $\mathsf{SPLP}(m, p, h)$, the procedure of *decoupling* is listed below.

- Solve subtask $\mathsf{S}_f(m, p, h)$. Let f be the result.
- Solve subtask $\mathsf{S}_c(m, p, h, f)$. Let c be the result.
- Take (f, c) as the synthesis result.

Theorem A.3 (Correctness of Decoupling). *Any* (f, c) *found by rule decoupling is a valid solution for* $\mathsf{SPLP}(m, p, h)$.

Besides, Theorem C.3, which demonstrates the effectiveness of *decoupling*, can also be generalized to lifting problems.

Theorem A.4. *For any task* $S_f(m, p, h)$*, a grammar* G_f *and a target program* $f^* \in G_f$*, the probability for a* $(\alpha, 0)$*-Occam solver to synthesize a program* f *different from* f^* *is negligible if (1) all related programs (*p*,* h *and programs in* G_f*) returns a constant number of scalar values; (2) all used arithmetic operators are linear; (3) the semantics of* p*,* h *and programs in* $G_f/\{f^*\}$ *are independently drawn from all functions mapping from the corresponding input space to the corresponding output space.*

A.3 Inductive Part

In the generalized version, S_c and S_f are still solved by *PolyGen* and *observational covering* respectively.

For task $\mathsf{S}_c(m, p, h, f)$, the following shows its specification in a point-free way, where $\mathbb{I}$ represents the input space.

$$\forall \bar{i} \in \mathbb{I}, p\ (m\ \bar{i}) = c\ (\mathsf{F}_m(p \triangle f)\ \bar{i})$$

Clearly, for each input $\bar{i}$, $(\mathsf{F}_m(p \triangle f)\ \bar{i}) \rightarrow (p\ (m\ \bar{i}))$ forms an input-output example for c. Therefore, the generalized version of S_c is still in the scope of *PolyGen*.

For task $\mathsf{S}_f(m, p, h)$, Lemma 4.6 (restated as Lemma A.5) still holds. Therefore, $\mathcal{O}_c$ is still applicable with all its properties preserved, as shown in Theorem A.6 and A.7

Lemma A.5. *For any task* $\mathsf{S}_f(m, p, h)$ *and a set* E *of examples, composed program* $\overline{f}$ *is observationally uncovered on* $E \Rightarrow \forall \overline{f'} \subseteq \overline{f}$*,* $\overline{f'}$ *is observationally uncovered on* E.

Theorem A.6. *Given task* $\mathsf{S}_f(m, p, h)$ *and a set* E *of examples, let* S *be the set of all valid composed programs. When* S *is non-empty,* $\mathcal{O}_c$ *always* ***terminates****. Besides, the program* $\overline{f^*}$ *synthesized by* $\mathcal{O}_c$ *always satisfies (1)* ***validity****:* $\overline{f^*} \in S$*, (2)* ***minimality****,* $\forall \overline{f} \in S, \neg\left(\overline{f} \prec_c \overline{f^*}\right)$.

Theorem A.7. *Synthesizer* $\mathcal{O}_c$ *(Algorithm 1) is a* $(1, 0)$*-Occam solver for* S_f.

B Appendix: Implementation

B.1 An Implementation of *AutoLifter* that Ensures Completeness

In this section, we introduce an implementation of *AutoLifter* that ensures completeness for lifting problems. Given task $\mathsf{LP}(M, p, h)$, the incomplete version of *AutoLifter* is comprised of four parts.

1. $\mathsf{LP}(M, p, h)$ is decomposed into a series of PLP tasks by applying rule *decomposition* recursively.
2. Each PLP task is further decomposed into $|M|$ SPLP tasks by applying rule *decomposition**.
3. For each task $\mathsf{SPLP}(m, p, h)$, $\mathcal{O}_m$ is invoked to synthesize a lifting scheme f from $\mathsf{S}_f(m, p, h)$.

4. Given lifting scheme f, an external solver, which is *PolyGen* in the incomplete implementation, is invoked to synthesize a combinator c form $\mathsf{S}_c(m, p, h, f)$.

To make *AutoLifter* complete, we should not miss any way to decompose $\mathsf{LP}(M, p, h)$ into PLP tasks. Note that in *decomposition*, subtask $\mathsf{LP}(M, f, h \triangle f)$ is related to f, which is synthesized from S_f in step 3. Therefore, we should not miss any solution to subtask S_f. To achieve this goal, a backtracking method is necessary to switch between different solutions to S_f and also different ways of the decomposition. Because $\mathcal{O}_c$ guarantees only the terminality for the first solution, but cannot tell whether all valid solutions have been found, the main challenge here is to determine when to switch to another search branch.

Algorithm 2 shows a complete version of *AutoLifter*, which solves the above challenge by iterating with a threshold *timeout* (Lines 35-40). In each turn, Algorithm 2 enumerates on solutions that can be found within *timeout* seconds.

Function `SPLPSolver` solves task $\mathsf{SPLP}(m, p, h)$ via rule *decoupling*. It returns all possible lifting schemes and the corresponding combinators that can be found within *timeout* seconds (Lines 1-10). Note that the choice of the combinator c does not affect the synthesis procedure of *AutoLifter*. Therefore, recording the first valid c for each possible f is enough. `SPLPSolver` firstly invokes `SFSolver` to find all solutions to $\mathsf{S}_f(m, p, h)$ with the time limit *timeout* (Line 3). `SPLPSolver` can be implemented as a basic enumerative solver that enumerates programs in G_f in order and returns those solutions found before timing out. For each found solution f, the external solver for S_f is invoked to solve $\mathsf{S}_c(m, p, h, f)$ using the remaining time (Line 4).

Function `PLPSolver` solves task $\mathsf{PLP}(M, p, h)$ via rule *decomposition**. It firstly solves each corresponding single partial lifting problem separately under the timeout (Line 12) and then returns those combinations of which the total time cost does not exceed *timeout* (Lines 14-22).

Function `LPSolver` solves task $\mathsf{LP}(M, p, h)$ via rule *decomposition* (Lines 23-34). First, it invokes `PLPSolver` to find solutions to the corresponding partial lifting problem with the timeout of *timeout* seconds (Lines 24). Then, it tries each solution recursively until a valid solution is found (Lines 28-32) or all solutions have been enumerated (Line 34).

Theorem B.1 shows the completeness of Algorithm 2.

Theorem B.1. *For any lifting problem* $\mathsf{LP}(M, p, h)$ *that has at least one solution, Algorithm 2 must terminate with a correct solution if* `SCSolver` *is complete, i.e., for any task* $\mathsf{S}_c(m, p, h, f)$ *that has at least one correct solution,* `SCSolver` *can always find a correct solution within finite time.*

The proof of this theorem is left to Appendix C.

B.2 Grammars

In this subsection, we supply the details to the default grammars G_f and G_c used in our implementation.

Algorithm 2: An implementation of *AutoLifter* that ensures completeness.

Input: A lifting problem $\mathsf{LP}(M = \{m_1, \ldots, m_n\}, p, h)$
Output: A valid solution $(f, c_1, \ldots, c_n)$.

```
1  Function SPLPSolver(m, p, h, timeout):
2      res ← [];
3      for (cost_f, f) ∈ SFSolver(m, p, h, timeout) do
4          solution ← SCSolver(m, p, h, f, timeout − cost_f);
5          if solution ≠ null then
6              (cost_c, c) ← solution;
7              res.Append((cost_f + cost_c, (f, c)))
8          end
9      end
10     return res;
11 Function PLPSolver(M, p, h, timeout):
12     ∀i ∈ [1, n], solutions_i ← SPLPSolver(m_i, p, h, timeout);
13     res ← [];
14     for solution_1 ∈ solutions_1, ..., solution_n ∈ solutions_n do
15         ∀i ∈ [1, n], (cost_i, (f_i, c_i)) ← solution_i;
16         total_cost ← Σ_{i=1}^{n} cost_i;
17         if total_cost ≤ cost then
18             f* ← f_1 △ ... △ f_n;  ∀i ∈ [1, n], c_i* ← c_i ∘ φ_i;
19             res.Append((total_cost, (f_1*, c_1*, ..., c_n*)));
20         end
21     end
22     return res;
23 Function LPSolver(M, p, h, timeout):
24     solutions ← PLPSolver(M, p, h, timeout);
25     for (cost, (f, c_1, ..., c_n)) ∈ solutions do
26         if f is null then return (f, c_1, ..., c_n);
27         solution ← LPSolver(M, f, h△f, timeout − cost);
28         if solution ≠ null then
29             (f', c_1', ..., c_n') ← solution;
30             ∀i ∈ [1, n], c_i* ← (c_i ∘ (φ_l × φ_l)) △ (c_i' ∘ (φ_r × φ_r));
31             return (f △ f', c_1*, ..., c_n*);
32         end
33     end
34     return null;
35 timeout ← 1;
36 while true do
37     timeout ← timeout × 2;
38     solution ← LPSolver(M, p, h, timeout);
39     if solution ≠ null then return solution;
40 end
```

Figure 4 shows the content of grammar G_f. Note that some operators in G_f are partial. For example, Head is defined only for non-empty lists. For convenience, we complete these operators with a dummy output $\perp$ and let the output of any operator on $\perp$ also be $\perp$.

Figure 5 shows the content of grammar G_c. Because the output of a lifting scheme can be the dummy value $\perp$, we

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

Figure 4. Grammar G_f used for solving task S_f.

Integer Expr	$N_{\mathbb{S}}$	$\rightarrow$	IntConst $\mid$ Inputs
		$\mid$	$N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid \bot$
		$\mid$	*ite* $N_{\mathbb{B}}$ $N_{\mathbb{Z}}$ $N_{\mathbb{Z}}$
Bool Expr	$N_{\mathbb{B}}$	$\rightarrow$	Inputs $\mid \neg N_{\mathbb{B}}$
		$\mid$	$N_{\mathbb{B}} \vee N_{\mathbb{B}} \mid N_{\mathbb{B}} \wedge N_{\mathbb{B}}$
		$\mid$	$N_{\mathbb{Z}} = N_{\mathbb{Z}} \mid N_{\mathbb{Z}} \leq N_{\mathbb{Z}}$
Binary Operator	$\oplus$	$\rightarrow$	$+ \mid - \mid \times \mid$ *div*

Figure 5. Grammar G_c used for solving task S_c.

also extend the semantics in G_c to support $\bot$ by setting the output of any operator on $\bot$ to $\bot$.

G_f and G_c satisfy the two assumptions discussed at the beginning of Section 4. First, all operators on Int and Bool in $G_f(G_c)$ are constant time. Second, for any program f in G_f, the output of f is a tuple of integers with a fixed size.

C Appendix: Proofs

In this section, we complete the proofs of the theorems in our paper.

C.1 Proofs for Section 4

Theorem C.1 (Theorem 4.3). *Any result found by rule decomposition is a valid solution for* $\mathsf{LP}(p, h)$.

Proof. This theorem is a special case of Theorem A.1. □

Theorem C.2 (Theorem 4.4). *Any* (f, c) *found by rule decoupling is a valid solution for* $\mathsf{PLP}(p, h)$.

Proof. This theorem is a special case of Theorem A.3. □

Theorem C.3. *For any task* $S_f(p, h)$*, a grammar* G_f *and a target program* $f^* \in G_f$*, the probability for a* $(\alpha, 0)$*-Occam solver to synthesize a program* f *different from* f^* *is negligible if (1) all related programs (*p*,* f *and programs in* G_f*) returns a constant number of scalar values; (2) all used arithmetic operators are linear; (3) the semantics of* p, h *and programs in* $G_f/\{f^*\}$ *are independently drawn from all functions mapping from the corresponding input space to the corresponding output space.*

Proof. This theorem is a special case of Theorem A.4. □

Lemma C.4 (Lemma 4.6). *Composed program* $\overline{f}$ *is uncovered on* $E \Rightarrow \forall \overline{f'} \subseteq \overline{f}$*,* $\overline{f'}$ *is uncovered on* E*, where* $\overline{f_1} \subseteq \overline{f_2}$ *if all lifting functions in* $\overline{f_1}$ *are in* $\overline{f_2}$.

Proof. This lemma is a special case of Lemma A.5. □

Theorem C.5 (Theorem 4.7). *Given task* $\mathrm{S}_f(p, h)$ *and a set* E *of examples, let* S *be the set of all valid composed programs. When* S *is non-empty,* $\mathcal{O}_c$ *always* ***terminates****. Besides, the program* $\overline{f^*}$ *synthesized by* $\mathcal{O}_c$ *always satisfies (1)* ***validity****:* $\overline{f^*} \in S$*, (2)* ***minimality****,* $\forall \overline{f} \in S, \neg\left(\overline{f} \prec_c \overline{f^*}\right)$.

Proof. This theorem is a special case of Theorem A.6. □

Theorem C.6 (Theorem 4.8). $\mathcal{O}_c$ *(Algorithm 1) is a* $(1, 0)$*-Occam solver.*

Proof. This theorem is a special case of Theorem A.7. □

Theorem C.7 (Theorem 4.9). *For any synthesis task* T*, any solver* $\mathcal{S}$ *available for task* T*, and any distribution* D *for examples, if the iterative algorithm terminates, the synthesized program* p *always satisfies the following formula.*

$$\forall \epsilon \in (2 \ln 2 / n_0, 1), \Pr\left[err_D(p) \geq \epsilon\right] \leq 4 \exp(-\epsilon n_0)$$

where $err_D(p)$ *represents the probability for* p *to violate an example drawn from distribution* D.

Moreover, the iterative algorithm always terminates when $\mathcal{S}$ *is an Occam solver.*

Proof. We start with the probability bound. Let $\mathcal{E}(j)$ be the random event that given $j \cdot n_0$ random examples, solver $\mathcal{S}$ returns a program of which the error rate is at least ϵ and the size is at most j. By the process of the iterative algorithms, we have the following inequality.

$$\Pr[err_{D,\overline{\varphi}}(p) \geq \epsilon] \leq \sum_{t=0}^{\infty} \Pr[\mathcal{E}(2^t)] \leq \sum_{j=1}^{\infty} \Pr[\mathcal{E}(j)]$$

When $\mathcal{E}(j)$ happens, there must a program satisfying that (1) its size is at most j, (2) its generalization error is at least ϵ, and (3) it satisfies all $n = j \cdot n_0$ random examples. Because size(p) is defined as the length of the binary representation of program p, there are at most 2^j programs satisfying the first condition. We denote these programs as $p_1, \ldots, p_m$ where $m \leq 2^j$. Then, we have the following inequalities.

$$
\begin{aligned}
&\Pr[\mathcal{E}(j)] \\
\le &\Pr_{\overline{i_1},\ldots,\overline{i_n}\sim D}\left[\exists k\in[1,m], err_{D,\overline{\varphi}}(p_k)\ge\epsilon \wedge \right. \\
&\qquad\qquad \left. p_k \text{ satisfies } \overline{i_1},\ldots,\overline{i_n}\right] \\
\le &\sum_{k=1}^{m}\Pr_{\overline{i_1},\ldots,\overline{i_n}\sim D}\left[err_{D,\overline{\varphi}}(p_k)\ge\epsilon \wedge p_k \text{ satisfies } \overline{i_1},\ldots,\overline{i_n}\right] \\
\le &\sum_{k=1}^{m}(1-\epsilon)^n \le 2^j\exp(-\epsilon\cdot j\cdot n_0)\le\exp(\ln 2-\epsilon n_0)^j
\end{aligned}
$$

Therefore, the generation error can be bounded.

$$
\begin{aligned}
\Pr[err_{D,\overline{\varphi}}(p)\ge\epsilon] &\le \sum_{j=1}^{\infty}\Pr[\mathcal{E}(j)] \\
&\le \sum_{j=1}^{\infty}\exp(\ln 2-\epsilon n_0)^j \\
&= \frac{2\exp(-\epsilon n_0)}{1-2\exp(-\epsilon n_0)} < 4\exp(-\epsilon n_0)
\end{aligned}
$$

The last inequality holds because $2\exp(-\epsilon n_0)$ is smaller than $1/2$ when $\epsilon > 2\ln 2/n_0$.

Then for terminality, suppose $\mathcal{S}$ is an $(\alpha,\beta)-$Occam solver for constant $\alpha\ge 1, 0\le\beta<1$, and the size of the smallest valid program is s. In the tth turn, the size of the program synthesized by $\mathcal{S}$ is at most $cs^\alpha 2^{t\beta}$ for some global constant c, and the threshold is 2^t. Consider the following derivation.

$$
cs^\alpha 2^{t\beta}\le 2^t \iff 2^{t(1-\beta)}\ge cs^\alpha \iff t\ge\left\lceil\frac{\ln c+\alpha\ln s}{\ln 2\cdot(1-\beta)}\right\rceil
$$

Therefore, when the number of turns is large enough, the threshold must be satisfied and thus the iterative algorithm must terminate when $\mathcal{S}$ is an Occam solver. □

C.2 Proofs for Appendix A

Theorem C.8 (Theorem A.1). *Any result found by decomposition is a valid solution for* $\mathsf{LP}(M,p,h)$.

Proof. Consider the procedure of *decomposition*, there are two possible cases. In the first case, $f = null$ in the first step.

$$
\begin{aligned}
&(null,c_1,\ldots,c_n) \text{ is valid for } \mathsf{PLP}(m,p,h)\\
\iff &p\circ m_i = c_i\circ \mathsf{F}_{m_i}(p\triangle null)\\
\iff &(p\circ m_i)\triangle null = (c_i\triangle null)\circ \mathsf{F}_{m_i}(p\triangle null)\\
\iff &(p\triangle null)\circ m_i = (c_i\triangle null)\circ \mathsf{F}_{m_i}(p\triangle null)\\
\iff &(null,c_1\triangle null,\ldots,c_n\triangle null) \text{ is valid for } \mathsf{LP}(M,p,h)
\end{aligned}
$$

In the second case, $f\neq null$ in the first step.

$$
\begin{aligned}
&(f,c_1,\ldots,c_n) \text{ is valid for } \mathsf{PLP}(M,p,h)\\
&(f',c_1',\ldots,c_n') \text{ is valid for } \mathsf{LP}(M,f,h\triangle f)\\
\iff &p\circ m_i = c_i\circ \mathsf{F}_{m_i}(p\triangle f)\\
&(f\triangle f')\circ m_i = c_i'\circ \mathsf{F}_{m_i}((h\triangle f)\triangle f')\\
\iff &p\circ m_i = c_i\circ \mathsf{F}_{m_i}(\varphi_l\circ(h\triangle(f\triangle f')))\\
&(f\triangle f')\circ m_i = c_i'\circ \mathsf{F}_{m_i}(\varphi_r\circ(h\triangle(f\triangle f')))\\
\iff &p\circ m_i = (c_i\circ \mathsf{F}_{m_i}\varphi_l)\circ \mathsf{F}_{m_i}(h\triangle(f\triangle f'))\\
&(f\triangle f')\circ m_i = (c_i'\circ \mathsf{F}_{m_i}\varphi_r)\circ \mathsf{F}_{m_i}(h\triangle(f\triangle f'))\\
\iff &(p\triangle(f\triangle f'))\circ m_i =\\
&((c_i\circ \mathsf{F}_{m_i}\varphi_l)\triangle(c_i'\circ \mathsf{F}_{m_i}\varphi_r))\circ \mathsf{F}_{m_i}(h\triangle(f\triangle f'))\\
\iff &(f\triangle f',c_1^*,\ldots,c_n^*) \text{ is valid for } \mathsf{LP}(M,p,h)
\end{aligned}
$$

□

Theorem C.9 (Theorem A.2). *Any result found by decomposition* is a valid solution for* $\mathsf{PLP}(M,p,h)$.

Proof.

$$
\begin{aligned}
&\forall i\in[1,n],(f_i,c_i) \text{ is valid for } \mathsf{SPLP}(m_i,p,h)\\
\iff &p\circ m_i = c_i\circ \mathsf{F}_{m_i}(p\triangle f_i)\\
\iff &p\circ m_i = c_i\circ \mathsf{F}_{m_i}(\varphi_i\circ(p\triangle(f_1\triangle\ldots\triangle f_n)))\\
\iff &p\circ m_i = (c_i\circ \mathsf{F}_{m_i}\varphi_i)\circ \mathsf{F}_{m_i}(p\triangle(f_1\triangle\ldots\triangle f_n))\\
\iff &(f_1\triangle\ldots\triangle f_n, c_1\circ \mathsf{F}_{m_1}\varphi_1,\ldots,c_n\circ \mathsf{F}_{m_n}\varphi_n) \text{ is valid}
\end{aligned}
$$

□

Theorem C.10 (Theorem A.3). *Any* (f,c) *found by rule decoupling is a valid solution for* $\mathsf{SPLP}(m,p,h)$.

Proof. This theorem is directly from the fact that c is a solution for $\mathsf{S}_c(m,p,h,f)$. □

Lemma C.11 (Completeness of Decoupling). *For any task* $\mathsf{SPLP}(m,p,h)$ *and any valid solution* f *for subtask* $\mathsf{S}_f(m,p,h)$, *subtask* $\mathsf{S}_c(m,p,h,f)$ *always has a solution if for any function* v *mapping from the output domain of* $\mathsf{F}_m(h\triangle f)$ *to the output domain of* $p\circ m$, *there is always a program in the program space of* c *that is semantically equivalent to* v.

Proof. For any solution f for $\mathsf{S}_f(m,p,h)$, define function v_f as the following, where v_0 is an arbitrary output.

$$
v_f\ x := \begin{cases} (p\circ m)\ \bar{i} & \exists\bar{i},(\mathsf{F}_m\ (h\triangle f))\ \bar{i} = x \\ v_0 & \forall\bar{i},(\mathsf{F}_m\ (h\triangle f))\ \bar{i}\neq x \end{cases}
$$

The following shows that $v_f\ x$ is a well-defined function.

$$
\begin{aligned}
&f \text{ is valid for } \mathsf{S}_f(m,p,h)\\
\iff &\mathsf{F}_m(h\triangle f)\ \bar{i} = \mathsf{F}_m(h\triangle f)\ \overline{i'} \to p\ (m\ \bar{i}) = p\ (m\ \overline{i'})\\
\iff &(v_f\ x) \text{ is unique for all } x
\end{aligned}
$$

By the condition, there exists a program c that is semantically equivalent to function v_f and thus satisfies $p\circ m = c\circ(\phi\ (h\triangle f))$. Therefore, c is valid for $\mathsf{S}_c(m,p,h,f)$. □

Theorem C.12 (Theorem A.4). *For any task $S_f(m,p,h)$, a grammar G_f and a target program $f^* \in G_f$, the probability for a $(\alpha,0)$-Occam solver to synthesize a program f different from f^* is negligible if (1) all related programs (p, h and programs in G_f) returns a constant number of scalar values; (2) all used arithmetic operators are linear; (3) the semantics of p, h and programs in $G_f/\{f^*\}$ are independently drawn from all functions mapping from the corresponding input space to the corresponding output space.*

Proof. The following is the specification of $\mathsf{S}_f(m,p,h)$.

$$\mathsf{F}_m(h \triangle f)\ \bar{i} = \mathsf{F}_m(h \triangle f)\ \overline{i'} \to p\ (m\ \bar{i}) = p\ (m\ \overline{i'})$$

Let $\mathbb{I}_n$ be a limited space containing all inputs where the size is at most n and the contents are integers in $[-n,n]$. The size of $\mathbb{I}_n$ is $(2n+1)^n = \exp(\Omega(n \ln n))$. In the following discussion, we assume that the domains of $\bar{i}$ and $\overline{i'}$ are $\mathbb{I}_n$.

Let f be an unwanted program in $G_f/\{f^*\}$. According to the first assumption, all of m,p,f outputs a constant number of scalar values. Let $n_o(g)$ be the number of scalar values in the output of program g. Clearly, $n_o(g) \le \text{size}(g)$.

According to the second assumption, all used arithmetic operators are linear. At this time, each operator in the program can only update each value via a linear expression on the existing values. Let m_c be the maximum absolute value of coefficients used by an operator. Then for any program g and any input $\bar{i}$ where the size is at most n_1 and the absolute value of the content is at most n_2, the absolute value of contents in $g\ \bar{i}$ is at most $n_2(cn_1)^{\text{size}(p)} = n_2 \exp(O(\text{size}(g)\ln n_1))$.

Let $\mathbb{O}_f$ be the range of $\mathsf{F}_m(h \triangle f)$ on $\mathbb{I}_n$. At this time, the size of the input is n, the contents in the input is in $[-n,n]$, and the size of $\mathsf{F}_m(h \triangle f)$ is $O(\text{size}(f))$. Therefore, $|\mathbb{O}_f| = (n \exp(O(\text{size}(f)\ln n)))^{n_o(\mathsf{F}_m(h\triangle f))}$. Because F_m and h are fixed, there exists constant c_o such that $n_o(\mathsf{F}_m(h\triangle f)) \le c_o n_o(f)$, and thus $|\mathbb{O}_f| = \exp(O(n_o(f)\text{size}(f)\ln n))$. Because $n_o(f)$ is at most $\text{size}(f)$, $|\mathbb{O}_f| = \exp(O(\text{size}(f)^2 \ln n))$.

Let $\mathbb{O}_p$ be the range of $p \circ m$ on $\mathbb{I}_n$. Similarly to $\mathbb{O}_f$, we have $|\mathbb{O}_f| = (n \exp(O(\text{size}(p\circ m)\ln n)))^{n_o(\mathsf{F}_m(p\circ m))}$. Because both p and m are fixed, $|\mathbb{O}_f| = \exp(O(\ln n))$.

Let $\overline{i_0}$ be any fixed input in $\mathbb{I}_n$, and $\bar{i}$ be another input in $\mathbb{I}_n/\{\overline{i_0}\}$. Under the third assumption, when the semantics are random, the probability for f to be invalid on example $(\overline{i_0}, \bar{i})$ is $a = (1 - |\mathbb{O}_p|^{-1})|\mathbb{O}_f|^{-1}$, which is at least $\frac{1}{2}|\mathbb{O}_f|^{-1}$ when $|\mathbb{O}_p| \ge 2$. Because the randomness comes from the semantics on input $\bar{i}$, the cases, where different $\bar{i}$ are used, are independent. As there are $|\mathbb{I}_n| - 1$ different choices of $\overline{i'}$, the following shows an upper bound on the probability for f to be valid on the input space $\mathbb{I}_n$.

$$(1-a)^{|\mathbb{I}_n|-1} \le \exp(-a(|\mathbb{I}_n|-1)) \le \exp(-(|\mathbb{I}_n|-1)/(2|\mathbb{O}_f|))$$

Now, consider the probability for a $(\alpha,0)$-Occam solver to synthesize an unwanted f. By the definition of Occam solvers, $\text{size}(f) \le c\,\text{size}(f^*)$, where c is some constant. Therefore, there are only $O(\exp(\text{size}(f^*)))$ different programs under such a size limit. Therefore, the following shows an upper bound on the probability for the Occam solver to return an unwanted result.

$$O(\exp(\text{size}(f^*))) \times \exp(-(|\mathbb{I}_n|-1)/(2|\mathbb{O}_f|))$$

Note that $|\mathbb{O}_f| = \exp(O(\text{size}(f)^2 \ln n))$ is significantly smaller than $|\mathbb{I}_n| = \exp(\Omega(n \ln n))$ when $n \to +\infty$. Therefore, such a probability becomes negligible when n is large enough. □

Lemma C.13 (Lemma A.5). *For any task $\mathsf{S}_f(m,p,h)$ and a set E of examples, composed program $\overline{f}$ is uncovered on $E \Rightarrow \forall \overline{f'} \subseteq \overline{f}$, $\overline{f'}$ is uncovered on E.*

Proof. Suppose $\overline{f}$ is uncovered on E and there is a composed program $\overline{f_1} \subseteq \overline{f}$ that is covered on E. At this time, there is a program $\overline{f_2} \prec_c \overline{f_1}$ and $E|_{\overline{f_1}} \subseteq E|_{\overline{f_2}}$.

We use $C(\overline{f})$ to denote the set of lifting functions used in $\overline{f}$. Let $\overline{f_3}$ be the composed program including lifting functions in $C(\overline{f})/C(\overline{f_1})$, and let $\overline{f'}$ be the composed program including lifting functions in $C(\overline{f_2}) \cup C(\overline{f_3})$. By the definition of $\prec_c$, it is easy to prove that $\overline{f'} \prec_c \overline{f}$. Meanwhile, we have the following inequality.

$$E|_{\overline{f}} = E|_{\overline{f_1}} \cup E|_{\overline{f_2}} \subseteq E|_{\overline{f_3}} \cup E|_{\overline{f_2}} = E|_{\overline{f'}}$$

Therefore, there is a composed program $\overline{f'}$ that is not only simpler than $\overline{f}$ under $\prec_c$ but also satisfies all examples satisfied by $\overline{f}$. Such a result conflicts with the fact that $\overline{f}$ is observationally uncovered on E. □

Theorem C.14 (Theorem A.6). *Given task $\mathsf{S}_f(m,p,h)$ and a set E of examples, let S be the set of all valid composed programs. When S is non-empty, $\mathcal{O}_c$ always **terminates**. Besides, the program $\overline{f^*}$ synthesized by $\mathcal{O}_c$ always satisfies (1) **validity**: $\overline{f^*} \in S$, (2) **minimality**, $\forall \overline{f} \in S, \neg\left(\overline{f} \prec_c \overline{f^*}\right)$.*

Proof. We start with the terminality. For each composed program $\overline{f} = [f_1, \ldots, f_k] \in S$, $f_1 \triangle \ldots \triangle f_k$ is inside G_f. Define set S' as the set of lifting schemes corresponding to composed programs in S, i.e., $\{f_1 \triangle \ldots \triangle f_k \mid (f_1, \ldots, f_k) \in S\}$. Clearly, lifting schemes in S' must be valid for $\mathsf{S}_f(m,p,h)$ on E. Because the enumerator $\mathcal{O}_e$ is complete, it can find a valid lifting scheme in finite time and thus $\mathcal{O}_c$ must terminate.

Then for validity, because $\mathcal{O}_c$ returns only when $\overline{f}$ is verified to be correct (Line 19), the result must be valid.

At last, we prove the minimality via the following claim.

- **Claim**: Each time when $\overline{f} = [f_1, \ldots, f_k]$ is added into *workingList*$[k]$ (Line 10), for all uncovered programs $\overline{f'} \prec_c \overline{f}$, $\overline{f'}$ must be included in either *workingList*$[k]$ or *minList*.

When this claim holds, suppose $\overline{f^*}$ is covered. There must a program $\overline{f}$ that is simpler than $\overline{f^*}$ under $\prec_c$ and satisfies all examples in E. By the claim, when $\overline{f^*}$ is added to $workingList[k]$, $\overline{f}$ is either inside $minList$ or $workingList[k]$.

- When $\overline{f}$ is inside $minList$, $\overline{f}$ must have been enumerated by the main loop in some previous turn. Therefore, $\mathcal{O}_c$ should have terminated before visiting $\overline{f^*}$, which contradicts with the fact that $\overline{f^*}$ is the result.
- When $\overline{f}$ is inside $workingList[k]$, according to Function `NextComposedProgram`, programs in $workingList[k]$ are returned in order. Therefore, $\overline{f}$ must also be visited by the main loop before $\overline{f^*}$. Similar with the previous case, at this time, $\overline{f}$ must be returned as the result instead of $\overline{f^*}$ and thus and thus a contradiction emerges.

Now we prove the claim by induction on the order of programs inserted into $workingList$. For composed program $\overline{g} = [g_1, \ldots, g_k]$ and any composed program $\overline{g'} = [g'_1, \ldots, g'_{k'}]$ that is simpler than $\overline{g}$ under $\prec_c$ and is observationally uncovered. There are three cases.

1. $k' < k$. By the definition of $\prec_c$, $[g_1 \ldots, g_{k-1}]$ is lexicographically no smaller than $[g'_1, \ldots, g'_{k'}]$. By the implementation of `InsertNewPrograms()` (Lines 9-14), program $[g_1, \ldots, g_{k-1}]$ is exactly the composed program $\overline{f}$ visited by the main loop. Therefore, we have $(\overline{f} = \overline{g'}) \vee (f \prec_c \overline{g'})$. For the former case, $\overline{f}$ has just been inserted into $minList$ (Line 10). For the latter case, the claim can be directly obtained from the induction hypothesis.
2. $k' = k$ and $[g_1, \ldots, g_{k-1}] \neq [g'_1, \ldots, g'_{k-1}]$. Similar with case (1), we know $[g_1, \ldots, g_{k-1}]$ is the composed program $\overline{f}$ visited by the main loop. Let composed program f' be $[g'_1, \ldots, g'_{k-1}]$. Then $\overline{f'}$ must be simpler than $\overline{f}$ by the definition of $\prec_c$. Therefore, by the induction hypothesis, $\overline{f'}$ must have been enumerated by the main loop in some previous turn, and in that turn, $\overline{g'}$ must have been added to $workingList[k]$. Therefore, the claim is obtained.
3. $k' = k$ and $[g_1, \ldots, g_{k-1}] = [g'_1, \ldots, g'_{k-1}]$. At this time, both $\overline{g}$ and $\overline{g'}$ will be added into the queue in the same invocation of `InsertNewPrograms()`. By the induction hypothesis, because $\mathcal{O}_e$ enumerates programs according to $\prec_s$, lifting functions in $minList[1]$ must be in the order of $\prec_s$. Therefore, $\overline{g'}$ will be inserted to $workingList[k]$ before $\overline{g}$ and thus the claim is obtained.

So far, we prove the claim, and thus prove the minimality. □

Theorem C.15 (Theorem A.7). *Synthesizer $\mathcal{O}_c$ (Algorithm 1) is a $(1, 0)$-Occam solver for S_f.*

Proof. Let $\overline{f} = [f_1, \ldots, f_k]$ be the composed program synthesized by $\mathcal{O}_c$ and $\overline{f^*}$ be the smallest valid program in G_f. Because (1) $\mathcal{O}_c$ only uses those programs enumerated by $\mathcal{O}_e$, (2) $\mathcal{O}_e$ enumerates programs from small to large, i.e., $\forall f_a, f_b \in G_f, \text{size}(f_a) < \text{size}(f_b)$ implies that $f_a \prec_s f_b$, we know that $\forall i \in [1, k], \text{size}(f_k) \leq \text{size}(f^*)$. Therefore, we have the following inequality.

$$\begin{aligned}\text{size}(f_1 \triangle \ldots \triangle f_k) &= \sum_{i=1}^{k} \text{size}(f_k) + (k-1) \times c \\ &\leq 2 \sum_{i=1}^{k} \text{size}(f_k) \\ &\leq 2k\text{size}(f^*) \leq 2n_c\text{size}(f^*)\end{aligned}$$

where c represents the binary bits used to express the operator $\triangle$. Because n_c is a constant, this inequality implies that $\mathcal{O}_c$ is a $(1, 0)$-Occam solver. □

C.3 Proofs for Appendix B

Theorem C.16 (Theorem B.1). *For any task $\mathtt{LP}(M, p, h)$ that has at least one solution, Algorithm 2 must terminate with a correct solution if `SCSolver` is complete, i.e., for any task $\mathsf{S}_c(m, p, h, f)$ that has at least one correct solution, `SCSolver` can always find a correct solution within finite time.*

Proof. Let $(f^*, c_1^*, \ldots, c_n^*)$ be any valid solution. Clearly, a solution to $\mathtt{LP}(M, p, h)$ will be found if `PLPSolver` could (1) find a solution using f^* as the lifting scheme for $\mathtt{PLP}(M, p, h)$, and (2) a solution using *null* as the lifting scheme for task $\mathtt{PLP}(M, f^*, h \triangle f^*)$.

Then, such two solutions will be found by `PLPSolver` is $\forall i \in [1, n]$, `SPLPSolver` could (1) find a solution using f^* as the lifting scheme for $\mathtt{SPLP}(m_i, p, h)$ and (2) find a solution using *null* as the lifting scheme for $\mathtt{SPLP}(m_i, f^*, h \triangle f^*)$.

These two conditions are equivalent to (1) `SFSolver` find some specific solution for $2n$ subtasks, and (2) `SCSolver` find a solution for $2n$ subtasks. By the definition of `SFSolver` and the completeness of `SCSolver`, all of these goals can be achieved within finite time, and thus their total time cost is also finite. Therefore, when *timeout* is iterated to a large enough number, a correct solution will be found . □

D Appendix: Algorithmic Tactics

In this subsection, we supply details on the four algorithmic tactics used in dataset $\mathcal{D}_L$.

D.1 Tactics for Longest Segment Problem

For the longest segment problem LSP, $\mathcal{D}_L$ involves 3 algorithmic tactics for predicates with different properties.

D.1.1 Tactic $\mathcal{A}_{l,1}$. The first $\mathcal{A}_{l,1}$ requires predicate b to be both *prefix-closed* and *overlap-closed*, where:

- Predicate b is *prefix-closed* if $b\ (l_1 +\!\!+ l_2) \rightarrow b\ l_1$.
- Predicate b is *overlap-closed* if the following is satisfied.

$$(len(l_2) > 0 \wedge b\ (l_1 +\!\!+ l_2) \wedge b\ (l_2 +\!\!+ l_3)) \rightarrow b\ (l_1 +\!\!+ l_2 +\!\!+ l_3)$$

Figure 6 shows the algorithmic template of $\mathcal{A}_{l,1}$. Function `lsp` is a single-pass function that consider each prefix of `A` in order. In the loop, three values `res`, `len` and `info` are stored.

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

Figure 6. The algorithmic template for tactic $\mathcal{A}_{l,1}$

- `res` represents the length of the longest valid segment.
- `len` represents the length of the longest valid suffix ls.
- `info` represents the output of $b \triangle f$ on ls.

Because b is prefix-closed and overlap-closed, each time when a new element is considered, the longest valid suffix must be $ls(A[0 \dots i-1]) +\!\!+ [A_i]$, $[A_i]$ or []. Therefore, `lsp` verifies these three choices and picks the first valid one among them (Lines 9-18). At this time, combinator c is used to quickly update `info` and verify whether $ls(A[0 \dots i-1]) +\!\!+ [A_i]$ is correct (Line 9).

To ensure the correctness, the following application condition must be satisfied. The corresponding synthesis task is $\mathsf{LP}(\{\lambda(l_1, a).\ l_1 +\!\!+ [a]\}, b)$.

$$(b \triangle f)\ (l +\!\!+ [a]) = c\ \Big((b \triangle f)\ l, a\Big)$$

D.1.2 Tactic $\mathcal{A}_{l,2}$. The second tactic $\mathcal{A}_{l,2}$ requires b to be *prefix-closed*, of which the template is shown as Figure 7.

Similar to tactic $\mathcal{A}_{l,1}$, $\mathcal{A}_{l,2}$ also calculates the longest valid suffix for each prefix of A (Lines 8-24). However, when b is only *prefix-closed*, we can only know that the longest valid suffix of prefix $A[0 \dots i]$ is equal to the longest valid suffix of $s = (ls\ (A[0 \dots i-1])) +\!\!+ [A_i]$. Therefore, $\mathcal{A}_{l,2}$ tries suffixes of s in order (Lines 8-22). Each time, $\mathcal{A}_{i,2}$ checks whether the current suffix is valid via combinator c_1 (Line 10). If it is not, $\mathcal{A}_{i,2}$ removes the first element via combinator c_2 (Line 20).

```
1  struct Info {
2    bool is_valid;
3    // Variables representing f l.
4  };
5  int lsp(int* A, int n){
6    int res = 0, len = 0;
7    Info info = (Info){/*b [], f []*/};
8    for (int i = 0; i < n; ++i) {
9      while (1) {
10       Info info2 = /*c1 info A[i]*/
11       if (info2.is_valid) {
12         info = info2;
13         len = len + 1;
14         break;
15       }
16       if (len == 0) {
17         info = (Info){/*b [], f []*/};
18         break;
19       }
20       info = /*c2 info A[i-len]*/;
21       len = len - 1;
22     }
23     res = max(res, len);
24   }
25   return res;
26 }
```

Figure 7. The algorithmic template for tactic $\mathcal{A}_{l,2}$

To ensure the correctness, the following application condition must be satisfied.

$$(b \triangle f)\ (l +\!\!+ [a]) = c_1\ \Big((b \triangle f)\ l, a\Big)$$
$$(head\ l = a) \rightarrow (b \triangle f)\ (tail\ l) = c_2\ \Big((b \triangle f)\ l, a\Big)$$

The corresponding synthesis task can be regarded as a limited version of $\mathsf{LP}(\{\lambda(l_1, a).\ (l_1 +\!\!+ [a]), \lambda(l_1, a).\ (tail\ l_1)\}, b)$, where the input (l_1, a) of the second modifier must satisfy $head\ l = a$. *AutoLifter* can be naturally extended to this task by filtering out invalid examples while sampling.

D.1.3 Tactic $\mathcal{A}_{l,3}$. The third tactic $\mathcal{A}_{l,3}$ does not have requirement on b, but is parameterized by a compare operator $R \in \{<, >, \leq, \geq\}$. The template of $\mathcal{A}_{l,3}$ is shown as Figure 8.

Tactic $\mathcal{A}_{i,3}$ uses a technique namely *segment partition*. Given an order R, the segment partition of a list $x[1 \dots n]$ is a series of segments $(r_0 = 0, r_1], (r_1, r_2], \dots, (r_{k-1}, r_k = n]$ satisfying (1) $\forall i \in [1, k], j \in (r_{i-1}, r_i), x_j\ R\ x_{r_i}$ and (2) $\forall i \in [2, k], \neg(x_{r_{i-1}}\ R\ x_{r_i})$. For convenience, we denote range (r_{i-1}, r_i) as the content of the ith segment $(r_{i-1}, r_i]$.

$\mathcal{A}_{i,3}$ maintains the segment partition for each prefix of A.

- `num` represents the number of segments in the partition.
- `rpos[i]` represents the value of r_i.
- `info[i]` records the outputs of p (the length of the longest valid segment) and f on the content of the ith segment.

```
1  struct Info{
2    int res; // Variable representing p l
3    // Variables representing f l
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 = {/*p [], f []*/};
10     while (num>0&&/*A[rpos[num]] R A[i]*/){
11       now=/*c info[num] A[rpos[num]] now*/;
12       --num;
13     }
14     num++; rpos[num]=i; info[num]=now;
15   }
16   Info now = {/*p [], f []*/};
17   for (int i = num; i > 0; i--) {
18     now = /*c info[i] A[rpos[i]] now*/;
19     merge(info[i], A[rpos[i]], now);
20   }
21   return now.res;
22 }
```

Figure 8. The algorithmic template for tactic $\mathcal{A}_{l,3}$

Each time, when a new element is inserted, $\mathcal{A}_{i,3}$ merges the last several segments together using combinator c to ensure that the remaining segments form a partition of the current prefix (Line 9-14). Then, after all elements are inserted, the segment partition of the whole list is obtained. $\mathcal{A}_{i,3}$ merges these segments together (Lines 16-20) and gets the result.

To ensure the correctness, the following application condition must be satisfied.

$$\Big((\forall a' \in l_1, a'Ra) \wedge (\forall a' \in l_2, \neg aRa')\Big) \rightarrow$$
$$(p \triangle f)\ (l_1 +\!\!+ [a] +\!\!+ l_2) = c\ \Big((p \triangle f)\ l_1, a, (p \triangle f)\ l_2\Big)$$

where p represents the program that calculates the length of the longest segment satisfying b. The synthesis task is a limited version of $\mathsf{LP}(\{\lambda(l_1, a, l_2), l_1 +\!\!+ [a] +\!\!+ l_2\}, p)$.

D.1.4 Guarantee. Similar to divide-and-conquer, for all above tactics, the efficiency of the synthesized program is guaranteed under the two assumptions made in Section 4.

Theorem D.1 (Efficiency on $\mathcal{A}_{l,i}$). *For any task* $\mathsf{LSP}(b)$*, let* p^* *be the program synthesized by applying* $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}$ *or* $\mathcal{A}_{l,3}$ *where the lifting scheme returns a constant number of scalar values. Then* p^* *is runs in linear time with respect to the length of the input list, under the assumption that any operator on scalar values is constant time.*

As discussed in Appendix B.2, the default grammars G_f and G_c used by *AutoLifter* satisfies both assumptions and thus the synthesis result of *AutoLifter* for tactics $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}$ and $\mathcal{A}_{l,3}$ are guaranteed to be efficient.

D.1.5 Usage. Similar to divide-and-conquer, given a black-box program p, the application of $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}$ and $\mathcal{A}_{l,3}$ can be fully automated via *AutoLifter*.

First, the semantics of predicate b required by $\mathcal{A}_{l,1}$ and $\mathcal{A}_{l,2}$ can be extracted from p as $b\ l := (p\ l = len\ l)$.

Second, given the semantics of predicate b, property *prefix-closed* and *overlap-closed* can be tested on random samples.

Third, there are at most 7 ways of applying these tactics: $\mathcal{A}_{l,1}, \mathcal{A}_{l,2}$, and $\mathcal{A}_{l,3}$ with four possible R. Therefore, we can enumerate on these 7 ways with a time limit, or directly tries them simultaneously in parallel.

D.2 Tactic for Range Update and Range Query

D.2.1 Tactic $\mathcal{A}_r$. Given task $\mathsf{RANGE}(h, u)$, let T_h, T_x and T_u be the types of the output of h, the elements in x, and the first parameter of u respectively. $\mathcal{A}_r$ requires the semantics of u to form a monoid. It requires one element $a_0 \in T_u$ and an operator $\otimes : T_u \times T_u \mapsto T_u$ satisfying the following formulas.

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

The template of tactic $\mathcal{A}_r$ is shown as Figure 9. $\mathcal{A}_r$ uses arrays `info` and `tag` to implement a segment tree.

- `info`[1] records the information on the root node, which corresponds to the whole list, i.e., range $[0, n-1]$.
- `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 function values of h and f on the segment corresponding to node k.
- 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 $u_{\texttt{tag}[k]}$, but such an update has not been applied to the subtree of node k yet.

There are several functions used in the template:

- `apply` deals with an update on all elements in the range corresponding to node `i` by updating `info`$[i]$ via combinator c_2 (Line 7) and updating the tag via $\otimes$ (Line 8).
- `pushdown` applies the tag on node `i` to its children (Lines 11), and clear the tag on node `i` (Line 12).
- `initialize` initializes the information for node `i` which corresponds to range $[l, r]$. It firstly recurses into two children (Lines 20-21) and then merges the sub-results together via combinator `c1` (Line 22).
- `upd` applies an update $([L, R], u_a)$ to node `i` which corresponds to range $[l, r]$. If $[l, r]$ does not overlap with $[L, R]$, the update will be ignored (Line 25). If $[l, r]$ is contained by $[L, R]$, the update will be performed via the lazy tag (Line 27). Otherwise, `upd` recurses into the two children (Lines 30-31) and merges the sub-results via `c2` (Line 32).
- `query` calculates a sub-result for query $[L, R]$ by considering elements in node `i` only. It is implemented similarly to function `upd`.

```
1  struct Info {
2    T_h res; // Variable representing h l.
3    // Variables representing f l.
4  }info[N];
5  T_u tag[N];
6  void apply(int i,T_u a){
7    info[i] = /*c2 info[i] a*/;
8    tag[i] = /*tag[i] ⊗ a*/;
9  }
10 void pushdown(int i){
11   apply(i*2, tag[i]); apply(i*2+1, tag[i]);
12   tag[i] = a_0;
13 }
14 void initialize(int i,T_x *A,int l,int r){
15   if (l == r) {
16     info[i] = /*h [A[l]], f [A[l]]*/;
17     return;
18   }
19   int mid = l + r >> 1;
20   initialize(i*2, A, l, mid);
21   initialize(i*2+1, A, mid+1, r);
22   info[i] = /*c1 info[i*2] info[i*2+1]*/;
23 }
24 void upd(int i,int l,int r,int L,int R,T_u a){
25   if (l > R || r < L) return;
26   if (l >= L && r <= R) {
27     apply(i, a); return;
28   }
29   int mid = l + r >> 1; pushdown(i);
30   upd(i*2, l, mid, L, R, a);
31   upd(i*2+1, mid+1, r, L, R, a);
32   info[i] = /*c1 info[i*2] info[i*2+1]*/;
33 }
34 Info query(int i,int l,int r,int L,int R){
35   if (l > R || r < L) return {/*h [], f []*/};
36   if (l >= L && r <= R) return info[i];
37   int mid = l + r >> 1; pushdown(i);
38   Info ql = query(i*2, l, mid, L, R);
39   Info qr = query(i*2+1, mid+1, r, L, R);
40   return /*c1 ql qr*/;
41 }
42 void range(int n,T_x *A,int m,Operator* op){
43   initialize(1, A, 0, n-1);
44   for (int i = 0; i < m; ++i) {
45     if (op[i].type == Update) {
46       upd(1,0,n-1,op[i].l,op[i].r,op[i].a);
47     } else {
48       Info r = query(1,0,n-1,op[i].l,op[i].r);
49       print(r.res);
50     }
51   }
52 }
```

Figure 9. The algorithmic template for tactic $\mathcal{A}_r$

To solve a Range task, $\mathcal{A}_r$ (1) initializes the segment tree via function `initialize` (Line 43), and then (2) invokes the corresponding functions for each operator (Lines 44-51).

To ensure the correctness, the following application condition must be satisfied.

$$(h \triangle f)\ (map\ u_a\ l) = c_1\ \Big((h \triangle f)\ l, a\Big)$$
$$(h \triangle f)\ (l_1 +\!\!+ l_2) = c_2\ \Big((h \triangle f)\ l_1, (h \triangle f)\ l_2\Big)$$

The synthesis task corresponding to this condition can be regarded as $\mathrm{LP}(\{\lambda(l_1, a).map\ u_a\ l_1, \lambda(l_1, l_2).\ l_1 +\!\!+ l_2\}, h)$.

D.2.2 Guarantee. For tactic $\mathcal{A}_r$, the efficiency of the synthesized program is still guaranteed under the two assumptions made in Section 4. Therefore, when the default grammars G_f, G_c are used, the efficiency of the result synthesized by *AutoLifter* for RANGE is guaranteed.

Theorem D.2 (Efficiency on $\mathcal{A}_r$). *For any task* RANGE(u, h), *let p^* be the program synthesized by applying $\mathcal{A}_r$ where the lifting scheme returns a constant number of scalar values. Then the time complexity of p^* is $O(n+m \log n)$, where n is the length of the input list and m is the number of operations, under the assumption that any operator on scalar values is constant time.*

D.2.3 Usage. When the semantics of the query function h and the update function u are given, the application of $\mathcal{A}_r$ can be fully automated via *AutoLifter*. Though element a_0 and operator $\oplus$ are required by $\mathcal{A}_r$, they can be synthesized from the semantics of u via an inductive synthesizer.

However, it is difficult to automatically apply $\mathcal{A}_r$ when only a black-box program p from RANGE(h, u) is given, because the semantics of p is not enough for us to either extract the semantics or get a complete specification for h and u. Because our paper focuses on the lifting problem, recovering the semantics of h and u from the semantics of p is out of our scope. Therefore, We leave this subtask to future work.

E Appendix: Evaluation

E.1 Extra Operators

As discussed in Section 7.3 and 7.4, for 9 tasks, we manually supply operators to *AutoLifter* under the enhanced setting. In this section, we report the details on these extra operators.

Task *atoi* in $\mathcal{D}_D$. The input program of task *atoi* is shown as Figure 10, which converts a list to an integer by regarding the list as a decimal string.

```
1 int atoi(int n, int *A) {
2   int res = 0;
3   for (int i = 0; i < n; ++i) {
4     res = res * 10 + A[i];
5   }
6   return res;
7 }
```

Figure 10. The input program of task *atoi*.

Under the enhanced setting, we supply operator $pos(x) := 10^x$ to grammar G_c.

Task *max_sum_between_ones* in $\mathcal{D}_D$. The input program of this task is shown as Figure 11, which calculates the maximum sum among segments that does not include number 1.

```
int max_sum_between_1s(int n, int *A) {
  int ms = 0, cs = 0;
  for (int i = 0; i < n; ++i) {
    cs = A[i] != 1 ? cs + A[i] : 0;
    ms = max(ms, cs);
  }
  return ms;
}
```

Figure 11. The input program of *max_sum_between_ones*.

Under the enhanced setting, we supply operator *pt1* to grammar G_f, where *pt1 l* is defined as the longest prefix of *l* that does not include 1 as an element.

Task *lis* in $\mathcal{D}_D$. The input program of task *lis* is shown as Figure 12, which calculates the length of the longest segment such that all elements are ordered.

```
int lis(int n, int *A) {
  int cl = 0, ml = 0, prev = A[0];
  for (int i = 1; i < n; ++i) {
    cl = prev < A[i] ? cl + 1 : 0;
    ml = max(ml, cl);
    prev = A[i];
  }
  return ml;
}
```

Figure 12. The input program of task *lis*.

Under the enhanced setting, we supply operator *lp* to grammar G_f. *lp* takes a list *l* and a binary compare operator $R \in \{<, >, \leq, \geq\}$ as the input, and returns the longest prefix of *l* that is ordered with respect to *R*.

Task *largest_peak* in $\mathcal{D}_D$. The input program here is shown as Figure 13, which calculates the maximum sum among those segments that all elements are positive.

```
int largest_peak(int n, int *A) {
  int cmo = 0, lpeak = 0;
  for (int i = 0; i < n; ++i) {
    cmo = A[i] > 0 ? cmo + A[i] : 0;
    lpeak = max(cmo, lpeak);
  }
  return lpeak;
}
```

Figure 13. The input program of task *largest_peak*.

Under the enhanced setting, we supply *lp*' to grammar G_f. *lp*' takes a list *l* and a predicate *b* as the input, and returns the longest prefix of *l* where *b* is satisfied by all elements.

Task *longest_reg* and *count_reg* in $\mathcal{D}_D$. There is a series tasks in $\mathcal{D}_D$ that performing regex matching on the list. These tasks can be divided into two categories:

- The input program of *longest_reg* calculates the length of the longest segment that matches a given regex.
- The input program of *count_reg* counts the number of segments that matches a given regex.

There are four such tasks on which *AutoLifter* fails if no extra operator is supplied: *count_1(0*)2*, *longest_1(0*)2*, *longest_(00)**, *longest_odd(0+1)**. Under the enhanced setting:

- We add operator *prefix_match* and *suffix_match* to grammar G_f for *count_1(0*)2*, *longest_1(0*)2*, *longest_odd(0+1)**. They take a list *l* and a regex *r* as the input, and return the longest prefix and suffix of *l* that matches *r* respectively. Besides, we also embed a sub-grammar for regex to G_f, which allows the *AutoLifter* to produce necessary regular expressions using $*$, | and concatenation.
- We add operator *mod2* to grammar G_c for *longest_(00)**and *longest_odd(0+1)**, because they require the combinator to tell the parity of an integer.

Benchmark *page21* in $\mathcal{D}_L$. The input program of *page21* is shown as Figure 14, which calculates the length of the longest segment satisfying that the leftmost element is the minimum and the rightmost element is the maximum.

```
int page21(int n, int *A) {
  int ans = 0;
  for (int i = 0; i < n; ++i) {
    int ma = -INF;
    for (int j = i; j < n; ++j) {
      if (A[j] < A[i]) break;
      ma = max(ma, A[j]);
      if (ma == A[j])
        ans = max(ans, j - i + 1);
      }
  }
  return ans;
}
```

Figure 14. The input program of task *page21*.

Under the enhanced setting, we supply operator *min_pos* to grammar G_f, which returns a list containing the positions of all prefix minimal in the input list.

E.2 Case Study

In this subsection, we complete the case study in Section 7.5.

Maximum Segment Product. The first problem is named as *maximum segment product (msp)* [Bird 1989], which is an advanced version of *mss*: Given list $l[1 \ldots n]$, the task is to select a segment *s* and maximize the product of values in *s*.

It is not easy to write a divide-and-conquer parallel program for this problem. According to the experience on solving *mss*, one may choose the maximum prefix/suffix product as the lifting functions. However, these two functions are not enough. It is counter-intuitive that the maximum segment product is also related to the **minimum** prefix/suffix product. This is because both the minimum suffix product and the maximum prefix product can be negative integers with a large absolute value, and thus their product flips back the sign, resulting in a large positive number.

This task foxes *Parsynt* as its transformation rules are not enough to extract these lifting functions, which are for the minimum, from the initial program, which is for the maximum. In contrast, by synthesizing from the semantics, *AutoLifter* successfully solve this task using 80.65s seconds. The solution found by *AutoLifter* is shown as the following.

$$f\ l \coloneqq \big(max\ (scanl\ \times\ l), max\ (scanl\ \times\ l), min\ (scanl\ \times\ l), min\ (scanl\ \times\ l), head\ (scanr\ \times\ l)\big)$$

$$msp\ (l_1 +\!\!+\ l_2) = \max\big(msp\ l_1, msp\ l_2, maxtp\ l_1 \times maxpp\ l_2, mintp\ l_1 \times minpp\ l_2\big)$$

The five lifting functions calculates the maximum prefix product (*maxpp*), the maximum tail product (*maxtp*), the minimum prefix product (*minpp*), the minimum tail product (*maxtp*) and the product of all elements (*prod*) respectively. For simplicity, we only report the partial combinator for *msp* here, and abbreviate *if-then-else* operators via max.

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

This problem is known to be difficult even for professional players in competitive programming. It has been set as a problem in 2020-2021 Winter Petrozavodsk Camp, which is a worldwide training camp representing the highest level of competitive programming, and the result is that only 26 out of 243 teams successfully solves this problem within 5 hours.

Tactic $\mathcal{A}_{l,3}$ with compare operator $>$ is applicable to solve this problem. The corresponding synthesis task is to find two programs f and c such that for any two lists l_1, l_2 and integer a satisfying that $\forall b \in l_1, a < b$ and $\forall b \in l_2, a \le b$:

$$(lsp \triangle f)\ (l_1 +\!\!+\ [a] +\!\!+\ l_2) = c\ \big((lsp \triangle f)\ l_1, a, (lsp \triangle f)\ l_2\big)$$

where *lsp* represents any correct program for this problem without considering the time complexity.

AutoLifter could find lifting functions $(len\ l, max\ l)$ and a correct combinator c using only 14.33 seconds. The structure of c is complex, and here we only explain the partial combinator c' for *lsp*. When $max\ l_1 \ge max\ l_2$, c' deals with the following 2 cases:

- When $a + max\ l_1 > len\ l_1 + 1$, the following expression shows the result of c'.

  $$\max(lsp\ l_2, \min(len\ l_1 + len\ l_2 + 1, a + max\ l_1 - 1))$$

  This is correct because there are only three possible cases for the longest valid segment: the longest valid segment s_1 in l_1, the longest valid segment s_2 in l_2, and the longest valid segment s_a contains element a.
  First, as $a + max\ l_1 > len\ l_1 + 1$, segment $l_1 +\!\!+ [a]$ is valid and thus s_1 is no longer than s_a.
  Second, s_a must be the prefix of $l_1 +\!\!+ [a] +\!\!+ l_2$ with length $\min(len\ l_1 + len\ l_2 + 1, a + max\ l_1 - 1)$, as this list has already included the largest element in the whole list.
  Therefore, the result must be the larger one between the length of s_2, $lsp\ l_1$, and the length of s_a.
- Otherwise, c' returns $\max(lsp\ p_1, lsp\ p_2)$ as the answer. This is correct because at this time, any valid segment s containing a must be no longer than $len\ l_1$. For any such segment s, let s' be any segment in l_1 that containing the largest element in l_1 and has the same length with s. Because $len\ s' = len\ s$, $min\ s' \ge min\ s$ and $max\ s' \ge max\ s$, s' must also be valid. At this time, because s' is inside l_1, it is no longer than the optimal segment in l_1, and thus s is also no longer than the optimal segment in l_1. Therefore, the result must be $\max(lsp\ l_1, lsp\ l_2)$ at this time.

For the case where $max\ l_1 < max\ l_2$, c' deals with it symmetrically. As we can see from this analysis, the correct combinator for this problem utilizes several tricky properties, and finding such a combinator is hard for a human user. In contrast, *AutoLifter* is able to solve this problem quickly.