Divide and Conquer Divide-and-Conquer

Black-Box Synthesis for Efficient Algorithms on Data Structures

ANONYMOUS AUTHOR(S)

Synthesizing efficient algorithms on data structures has long been a difficult task. In this paper, we observe that the core tasks of synthesizing many classes of algorithms on data structures, including parallelization, incrementalization, and efficient search trees, can be viewed in general as synthesizing divide-and-conquer (D&C) algorithms by using different ways to divide the input. Motivated by this observation, we propose a novel general approach, *AutoLifter*, for synthesizing these D&C algorithms in a broad sense, which is captured as the lifting problem.

We use divide-and-conquer to synthesize efficient divide-and-conquer algorithms efficiently. It uses two techniques, decomposition and decoupling, to divide the lifting problem into subtasks. To solve those complex subtasks, we use weaker but simpler tasks to approximate them and prove the Occam-approximation theorem to guarantee the performance of this approximation. Besides, we also propose a pruning strategy, observational covering, to implement efficient synthesizers for the subtasks.

We evaluate *AutoLifter* on 57 programming tasks related to five different classes of D&C algorithms. The results show that *AutoLifter* is able to solve 56/57 tasks with 8.38 seconds on average, and achieve competitive, or even better, performance compared with an existing specialized synthesizer that requires more input.

1 INTRODUCTION

Search-based program synthesis [Alur et al. 2018] has been a research focus in recent decades and found applications in a variety of problems. However, it remains as a big challenge to synthesize efficient algorithms on data structures. To see this, consider the well-known maximum segment sum problem (*mss*) [Bird 1989b], which is to compute, given a list of integers, the maximum of the sums of the contiguous segments of that list. One can come up with the following straightforward specification in Haskell:

$$mss = maximum\ .\ map\ sum\ .\ segments$$

which, given a list, enumerates all its segments, computes the sum for each segment, and returns the maximum of all the sums. This is a cubic algorithm, which is inefficient. How can we automatically synthesize a linear-time sequential algorithm for solving it? Moreover, how can we automatically synthesize an efficient parallel algorithm for solving it?

To answer these questions, we may impose some reasonable constraints on the shape of the synthesis programs, and thus reformulate this challenging synthesis problem as finding general divide-and-conquer (D&C) algorithms that are efficient. By "D&C", we mean that we divide the input data structure into instances of the same type, (recursively) solve the problem on each of the instances, and combine the solutions. For instance, for a nonempty list xs, we may divide it into $y : ys$, where y is its head element and ys is the tail part of the list, conquer the problem on smaller list ys, and combine this result with y to produce the final result. There are many other ways to divide a list. For example, we may divide a list xs into a concatenation of two lists $ys +\!\!+ zs$, or into concatenation of three parts $ys +\!\!+ [a] +\!\!+ zs$, where a is the head element, ys contains those elements that are smaller than a, and zs contains the rest of the elements. In fact, as will be seen in Section 3, this D&C has a broad meaning, covering a wide variety of algorithm classes where some are traditionally not considered as D&C, because of powerful and flexible ways of dividing.

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

Now what do we mean by "efficient" under the D&C computation scheme? We may define that a D&C program is efficient, if there are neither unnecessary intermediate data structures that are produced and consumed during the computation, nor multiple traversals of the same data structure for computing different results from the same data. In fact, this definition follows directly from the two most general optimization strategies, fusion and tupling, which have been intensively studied for optimization of functional and logic programs [Pettorossi and Proietti 1996].

However, not all programming problems can be directly described in the D&C manner. For *mss*, there is neither a binary operator $\oplus$ satisfying

$$mss\ (y : ys) = y \oplus mss\ ys$$

nor a binary operator $\otimes$ satisfying

$$mss\ (ys +\!\!+ zs) = mss\ ys \otimes mss\ zs.$$

To implement *mss* in the D&C manner, we need to compute some auxiliary values along with the results. For *mss*, in addition to compute the result of the maximum segment sum, if we compute the maximum prefix sum (*mps*), the maximum tail-segment sum (*mts*) and the sum of the list (*sum*), by defining mss' as

$$mss'\ xs = (mss\ xs, mps\ xs, mts\ xs, sum\ xs)$$

then we can compute *mss* through mss' by

$$mss\ xs = fst\ (mss'\ xs)$$

whereas mss' can be implemented in the D&C manner; we can find a combine operator $\odot$ satisfying

$$mss'\ (ys +\!\!+ zs) = mss'\ ys \odot mss'\ zs.$$

As will be discussed in Section 2, it is hard to find out what auxiliary values are needed.

In this paper, we propose a new general method for synthesizing efficient D&C algorithms through the above procedure with a system *AutoLifter*. We call such a synthesis problem a *lifting problem* because the major task is to find auxiliary values that lift the original result to make the combine operator $\odot$ computable. The main challenge for solving a lifting problem is the scale of the synthesis result. Modern program synthesis algorithms are typically search-based, using a search algorithm to explore the syntactic space of programs. Since the program space grows exponentially with the size of the largest program, it is difficult for them to scale to large programs. However, a solution to a lifting problem is usually large, consisting of tens of operators and beyond the capability of the state-of-the-art general-purpose synthesizers.

To address this scalability challenge, we propose to *divide* the synthesis target into a sequence of subparts, each tractable by a search-based synthesizer, and synthesizes these subparts one by one. In other words, we apply D&C to the synthesis of D&C. Since a precise specification for each subpart may be difficult for direct synthesis, our approach derives weak specifications, and the synthesis may fail if a subpart is synthesized incorrectly. We further prove the Occam-approximation theorem, which gives a sufficient condition of bounding the error rate of synthesis using a weak specification, and prove that *AutoLifter* satisfies the condition.

To make our synthesis method generic and reduce the burden of specifying a programming task, we study the *black-box* synthesis for the lifting problem, in the sense that the specification of the problem (e.g., the inefficient *mss* implementation) and the dividing method are both given as black-box programs. In this way, the user is free to specify the problem in any algorithm and any programming language, and our approach can be applied to different classes of D&C algorithms by using different dividing methods.

To show power of our approach, we have implemented *AutoLifter* as five synthesizers, each for a class of D&C algorithms, including automatic parallelization [Farzan and Nicolet 2017, 2021b;

Morita et al. 2007; Raychev et al. 2015], segment trees [Lau and Ritossa 2021], and three classes of greedy algorithms for the longest segment problem [Zantema 1992]. We collect a dataset of 57 programming tasks, including 35 for parallelization and 22 for the other four algorithm classes, from existing benchmarks [Bird 1989a; Farzan and Nicolet 2017, 2021b; Morita et al. 2007], existing publications on formalizing algorithms [Zantema 1992], and an online contest platform for competitive programming (codeforces.com). We apply the respective synthesizers to the tasks. The results show that *AutoLifter* is able to solve 56 out of 57 tasks with an average time cost of 8.38 seconds, and achieves competitive, or even better, performance compared with an existing specialized approach for automatic parallelization that requires more input from the user. 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 makes the following contributions:

- Defining the *lifting problem*, which captures many classes of efficient algorithms on data structures.
- Proving the *Occam-approximation theorem*, which help bound the error rate of synthesis using a weak specification.
- Proposing *AutoLifter*, an efficient solver that divides and conquers the lifting problem guided by the above theorem.
- Conducting an *empirical evaluation*, which shows the effectiveness of *AutoLifter*.

2 OVERVIEW

In this section, we give a brief overview about our synthesis idea for synthesizing a specific class of D&C algorithms for efficient parallel computation (also known as list homomorphism) with two running examples.

2.1 Running Examples

We start with a simple example of calculating the average value of a list of integers, which is specified as

$$average\ xs = div\ (sum\ xs)\ (length\ xs).$$

The problem is to find a D&C algorithm where the input list xs is divided into a concatenation of two sublists $ys +\!\!+ zs$. In other words, we wish to synthesize a binary operator $\oplus$ such that the following holds.

$$average\ (ys +\!\!+ zs) = average\ ys \oplus average\ zs$$

However, such $\oplus$ does not exist, because we cannot calculate the average value of the whole list from the average values of the two arbitrary sublists whose concatenation forms the original input list. Fortunately, our synthesis method can automatically find the following function

$$f_{avg}\ xs = (sum\ xs, length\ xs)$$

to compute auxiliary values from the sublists, such that the new function

$$average'\ xs = (average\ xs, f_{avg}\ xs)$$

can be defined as a D&C algorithm for xs being divided into $ys +\!\!+ zs$:

$$\begin{aligned} &average'(ys +\!\!+ zs) = average'\ ys \oplus average'\ zs \\ &\quad \textbf{where}\ (a_L, (s_L, l_L)) \oplus (a_R, (s_R, l_R)) = (div\ (s_L + s_R)\ (l_L + l_R), (s_L + s_R, l_L + l_R)) \end{aligned}$$

It is worth noting that the above procedure going from *average* to *average′* is often called *parallelization* [Farzan and Nicolet 2017, 2021b; Morita et al. 2007; Raychev et al. 2015], because

the obtained D&C algorithm is suitable for parallel computation; we may divide a list into half, perform computation on each part in parallel, and finally combine the results with $\oplus$.

As we can see from the *average* example, deriving a D&C algorithm for parallel computation is not straightforward as we need to discover auxiliary values. This difficulty is not prominent in this example because the auxiliary values are also produced during the computation of the original program. As the second example, let us recall the maximum segment sum problem in the introduction. Our synthesizer can smartly find the following auxiliary function f_{mss} defined by

$$f_{mss}\ xs = (maximum\ (scanl\ (+)\ 0\ xs),\ maximum\ (scanr\ (+)\ 0\ xs),\ sum\ xs)$$

for computing three auxiliary values, namely the maximum prefix sum (mps), the maximum tail-segment sum (mts), and the sum of the whole list, respectively. In addition, it can derive the following $\oplus$ for combining the results:

$$\begin{aligned}&(mss_L, (mps_L, mts_L, sum_L)) \oplus (mss_R, (mps_R, mts_R, sum_R))\\ &\quad = (maximum\ [mss_L, mss_R, mts_L + mps_R],\\ &\qquad (max\ mps_L\ (sum_L + mps_R), max\ (mts_L + sum_R)\ mts_R, sum_L + sum_R))\end{aligned}$$

As we can see from the *mss* example, the function f_{mss} for calculating the three values is very different from original program and is not easy for the developer to derive.

2.2 Abstraction as a Lifting Problem

As seen in the above, our specific synthesis task here is to find f and $\oplus$ for a given function p such that the tupled function $p' = p \vartriangle f$, where $(f \vartriangle g)\ x = (f\ x, g\ x)$, can be defined as follows[1]:

$$p'(xs +\!\!+ ys) = p'\ xs \oplus p'\ ys \tag{1}$$

where $\oplus$ is a binary operator. Here, p' is often called a list homomorphism because it preserves the list concatenation operation $+\!\!+$ as the $\oplus$ operation in its codomain. In both of our examples, the original program p cannot preserve the list concatenation operation as any operation in its codomain, and the function f lifts p as a list homomorphism p'. In this sense, function f is also called a *lifting function*. In this paper we refer such a synthesis problem as a *lifting problem*. In this section we only consider the preservation of list concatenation, and later we shall generalize lifting problems to preserve different operations, where each represents a different way of dividing the input data structure, and possibly different classes of algorithms.

Solving a lifting problem is challenging. On the one hand, the scale of f and $\oplus$ may be large. In the *mss* example, the two functions use 28 operators in total, 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 with a time limit of one hour. On the other hand, there are two functions to be synthesized, and they intertwine with each other in a relation specification. This renders many optimizations [Balog et al. 2017; Feser et al. 2015; Osera and Zdancewic 2015; Rolim et al. 2017] based on input-output examples impossible, while existing relational synthesizers [Alur et al. 2013; Wang et al. 2018] also do not scale to such large programs.

2.3 Decoupling and Occam-Approximation Theorem

Now we discuss our key idea "decoupling" for solving the lifting problem, which separates the synthesis of f as an independent subproblem. After f is synthesized, the specification of the lifting

[1]For readability, in this paper, we mark those unknown functions (i.e., synthesis targets) as blue and mark those first-order variables bounded by $\forall$ as green.

problem can be instantiated as a specification on only $\oplus$, which can be synthesized using existing synthesizers.

A precise specification over f is to introduce an existential quantifier on $\oplus$ for the original specification, as shown below. Here $G_\oplus$ is the syntactic space for $\oplus$.

$$\exists \oplus \in G_\oplus, p'(xs +\!\!+ ys) = (p'\ xs) \oplus (p'\ ys) \text{ where } p' = p \vartriangle f \tag{2}$$

However, it is difficult to use this specification directly in synthesis: given f, we cannot even verify or validate its correctness because of the second-order existential quantification over $\oplus$.

To solve this problem, we derive a weaker specification over f without existential quantification to approximate the above precise specification. We notice that $\oplus$ is a function, and thus the same input must lead to the same output. Therefore, we have the following specification for any lists xs, ys, xs', ys', which involves only f.

$$p'\ xs = p'\ xs' \wedge p'\ ys = p'\ ys' \longrightarrow p'\ (xs +\!\!+ ys) = p'\ (xs' +\!\!+ ys') \text{ where } p' = p \vartriangle f \tag{3}$$

This specification is weaker, because it assumes any $\oplus$, while the precise specification (Formula 2) considers only $\oplus$ in $G_\oplus$. It is possible that we can synthesize f but do not have a corresponding $\oplus$ in $G_\oplus$. For example, suppose the given program p is *average* and language $G_\oplus$ has a multiplication operator but do not have the operator for the square root. The function

$$f\ xs = (sum\ xs * sum\ xs, length\ xs)$$

satisfies the above specification but may not have the corresponding $\oplus$ when no operator for the square root is included. Therefore, we need to ensure that any f synthesized using the above weaker specification has a high probability to be valid for the precise specification.

However, analyzing the above probability is not easy. Given a function f, it is even difficult to know whether it satisfies the precise specification or not. To fill this gap, we conduct a theoretical analysis on the error rate of synthesizing using a weak specification to approximate a strong specification. Our analysis is based on the concept of Occam solver, a special class of synthesizers defined by Ji et al. [2021]. An α-Occam solver ensures that the returned program has the size of $O(size(f)^\alpha)$, for any program f that satisfy the specification. We also define the synthesis domain of a synthesizer for a program f^* as all possible programs it may return for any specification where f^* is valid. As a result, the synthesis domain of an Occam solver is bounded by the size of f^*.

A key result of our analysis is the Occam-approximation theorem. This theorem shows that, if we would like to bound the error rate using a weak specification, it is sufficient to satisfy the following two conditions: (1) the solver is an Occam solver, and (2) for valid program f^*, the probability of all other programs in the synthesis domain to satisfy the weak specification is small. These conditions depend only on the weak specification, and thus we can avoid analyzing the more difficult precise specification.

Guided by this theorem, we design a bottom-up enumerative synthesizer that enumerates the programs from smaller to bigger. It is easy to see that this solver is a 1-Occam solver. This solver also satisfies the second condition. In our example, the aforementioned incorrect program would not be generated because they are not in the synthesis domain. Other pairs of common functions on list in the synthesis domain, such as (max, min), cannot satisfy the specification. Intuitively, this is because f is a compressing function: it returns a finite number of values from an arbitrarily large list, and thus for any random function p and f, there is a high probability to find some list such that the above specification does not hold.

Please note that our approach is sound, and we would not return an incorrect program even if an incorrect f is synthesized. This is because the synthesis of $\oplus$ uses the precise specification and would only fail when an incorrect f is synthesized.

2.4 Decomposition

After decoupling, the individual f and $\oplus$ may still be too large to synthesize. Even in the trivial example of calculating the average value, the $\oplus$ function has 7 operators, which is already difficult for many existing synthesizers to handle. Therefore, we propose a decomposition method to further separate these functions into subparts.

In the example of *average'*, we notice that $\oplus$ can be divided into two subparts serving different purposes. The first subpart, a, creates the combined value for p and the second subpart, (s,l), creates the combined value for f. Therefore, we can synthesize them one by one.

In *AutoLifter*, this decomposition happens before the decoupling of f and $\oplus$. To synthesize only the first subpart, we obtain the following specification.

$$p\ (xs +\!\!+ ys) = ((p \vartriangleleft f_1)\ xs) \oplus_1 ((p \vartriangleleft f_1)\ ys) \tag{4}$$

Here $\oplus_1$ refers to the first subpart of $\oplus$, while f_1 refers to the auxiliary values for p, which is equal to f in the example. Please note that this specification has a similar structure to the original one, and thus f_1 and $\oplus_1$ can be synthesized through decoupling.

After f_1 and $\oplus_1$ are synthesized, we obtain the following specification for the second subpart.

$$f_1\ (xs +\!\!+ ys) = ((p \vartriangleleft f_1 \vartriangleleft f_2)\ xs) \oplus_2 ((p \vartriangleleft f_1 \vartriangleleft f_2)\ ys)$$

Here f_2 is the auxiliary values potentially needed for f_1. This specification has the same form as Formula 4 and thus f_2 and $\oplus_2$ be synthesized in the same way. In this example, $f_2 = ()$ as no auxiliary value is needed, and the synthesis process ends.

In more complex cases, the synthesis may take more steps, where each step provides the auxiliary values and the combination operator for the previous step. In our second example of mss, the decomposition process consists of three steps, as shown below.

$f_1 = mps \vartriangleleft mts$	$(mss_L, (mps_L, mts_L)) \oplus_1 (mss_R, (mps_R, mts_R)) =$ $maximum\ [mss_L, mss_R, mts_L + mps_R]$
$f_2 = sum$	$((mss_L, (mps_L, mts_L)), sum_L) \oplus_2 ((mss_R, (mps_R, mts_R)), sum_R) =$ $(max\ mps_L\ (sum_L + mps_R), max\ (mts_L + sum_R)\ mts_R)$
$f_3 = ()$	$((mss_L, (mps_L, mts_L), sum_L), ()) \oplus_3 ((mss_R, (mps_R, mts_R), sum_R), ()) =$ $sum_L + sum_R$

The above specifications for the subparts are still weak specifications. Let us consider Formula 4 and our first running example. If we synthesize f_1 as

$$f_1\ xs = (sum\ xs * length\ xs, length\ xs * length\ xs)$$

then $\oplus_1$ remains the same, but $\oplus_2$ cannot be synthesized if we do not have the operator for calculating the square root. Nevertheless, our approach still bounds the error rate and the error rate can also be reasoned via the Occam-approximation theorem.

2.5 Optimization: Observational Covering

As discussed before, *AutoLifter* uses a bottom-up enumerative synthesizer to synthesize f from the weaker specification generated by decoupling. A potential problem here is the scalability. Though multiple techniques have been proposed to scale up enumerative synthesis, the scalability may not be enough for many D&C algorithms. To deal with issue, we propose a novel pruning technique, *observational covering*, based on the structure of f.

We shall illustrate observational covering on the first subtask generated by decomposition (Formula 4) with our first example of calculating the average. At this time, the weak specification

generated by decoupling is as follows, where p is *average*.

$$(p \vartriangle f_1)\ xs = (p \vartriangle f_1)\ xs' \wedge (p \vartriangle f_1)\ ys = (p \vartriangle f_1)\ ys' \longrightarrow p\ (xs +\!\!+ ys) = p\ (xs' +\!\!+ ys') \tag{5}$$

We assume the synthesizer is implemented in the CEGIS framework [Solar-Lezama et al. 2006], i.e., the goal is to satisfy a set of examples. In our case, an example possibly generated by CEGIS is two pairs of lists, (xs, ys) and (xs', ys'), whose equality on *average* is not preserved by concatenation, i.e., $p\ xs = p\ xs' \wedge p\ ys = p\ ys'$ but $p\ (xs +\!\!+ ys) \neq p\ (xs' +\!\!+ ys')$. To satisfy this example, f_1 should violate the premise of the weaker specification by outputting differently on the two pairs of lists, i.e., $f_1\ xs \neq f_1\ xs'$ or $f_1\ ys \neq f_1\ ys'$.

We observe that f_1 is formed by a tuple of functions, i.e., *sum* and *length*, where each provides an auxiliary value. Then for each example, it is sufficient for one function f_i in this tuple to produce different values on the two pairs to make the two pairs unequal. In such a case, we say $f_{1,i}$ covers this example. Therefore, if a function f_a only covers a subset of examples that function f_b covers, f_a is subsumed by f_b and should not be selected as a component in the tuple.

For example, consider the following set of two examples.

- $p\ [1,1] = p\ [1] = 1, p\ [3] = p\ [3,3] = 3$, but $p\ [1,1,3] = 1 \neq p\ [1,3,3] = 2$.
- $p\ [1,1] = p\ [1] = 1, p\ [3] = p\ [2,4] = 3$, but $p\ [1,1,3] = 1 \neq p\ [1,2,4] = 2$.

Function *max* covers the first but not the second, and is subsumed by *length*, which covers both.

In a bottom-up enumerative synthesis process, we enumerate the programs from small to large by combined enumerated programs with language constructs. Based on the above observation, when selecting enumerated programs to form the tuple of f_1, we select only those that are not covered by other programs. In this way, significant search space is pruned off.

3 LIFTING PROBLEM

In this section, we give the formal definition for the lifting problems for the synthesis of efficient D&C algorithms (programs).

Notations. In this paper we regard a type as a set and use the two terms interchangeably. We further distinguish two special sets: scalar types (e.g., `Int`, `Bool`, and their tuples) and recursive data structure types (e.g., `List` and `Tree`). We assume all scalar types are constant-size, while the size of recursive data types can be arbitrarily large. We also assume any function whose domain is a scalar type is implemented as a constant-time algorithm. While this assumption does not hold in general (e.g., calculating the largest prime number smaller than n), it is not an issue in synthesizing algorithms over data structures as we usually only apply constant-time operations on scalars. We use uppercase letters such as A, B to denote types, lowercase letters such as f, g to denote functions, and overline letters such as $\overline{a}, \overline{b}$ to represent vectors. We use T^n to represent n-arity product $T \times \cdots \times T$ for set T, and f^n to apply function f to each component in an n-tuple.

$$f^n\ (x_1, \ldots, x_n) = (f\ x_1, \ldots, f\ x_n)$$

Lifting Problem. In Section 2, we have seen a lifting problem which is to lift the input program as a list homomorphism. Here, we give a more general definition.

Definition 3.1 (Lifting Problem). Given function p with domain A, an operator $\mu : C \times A^n \to A$, and two grammars G_f, G_τ, lifting problem $\mathsf{LP}(p, \mu, G_f, G_\tau)$ is to find a lifting function $f \in G_f$ and an operator $\tau \in G_\tau$ satisfying the formula below for any x in C and $\overline{a} \in A^n$.

$$(p \vartriangle f)\ (\mu\ (x, \overline{a})) = \tau\ \left(x, (p \vartriangle f)^n\ \overline{a}\right) \tag{6}$$

Figure 1 shows the commutative diagram of Formula 6, where blue arrows correspond to the left-hand side, and green arrows correspond to the right-hand side. As we can see from the figure,

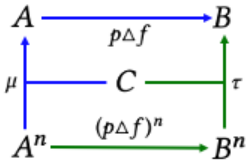

Fig. 1. The commutative diagram corresponding to the lifting problem.

$(p \triangle f)$ is a homomorphism that preserves the operation μ in the domain A as operation τ in the domain B.

In the sense of D&C synthesis, domain A is the domain of the input data structure such as lists, p is the input program specifying the problem, and μ is an operator representing how we divide the input. In the case of automatic parallelization, μ is list concatenation in the sense that the list is divided as two sub-lists that concatenates to the original one. In other classes of algorithms, μ can be different operators. C is a type includes complementary inputs of μ that are not of the original data structure type. In the case of list concatenation where complementary inputs are not needed, C is a singleton set with only one member. In other dividing methods, such as divide a list as a head and a tail, C captures the head value which is not a list. The goal of the synthesis is to find f that provides auxiliary values such that τ (corresponding to $\oplus$ in the example) can combine the results on the sub-lists into the result of the whole list. Here B is the domain of the expected results and the auxiliary values. We also assume that grammar G_f contains a constant function *null* that maps anything to the unit constant () to represent the case where p is already a homomorphism without a lifting function.

For simplicity, in this paper, we assume that the two grammars G_f and G_τ are implicitly given and thus abbreviate a lifting problem $\mathsf{LP}(p, \mu)$.

Lifting Problem with Efficiency Guarantee. The general definition of the lifting problem does not ensure the efficiency of the synthesized program. In an extreme case, function f directly copies the input list as an auxiliary value, and $\oplus$ applies the original program p to the auxiliary value. Such a synthesized result is functionally correct but only has a worse complexity than p. To ensure the efficiency of the synthesized algorithm, we may add the following condition to the lifting problem.

Definition 3.2 (Lifting Problem with Efficiency Guarantee). A lifting problem $\mathsf{LP}(p, \mu)$ has *efficiency guarantee* if the type C, the range of p, and the range of each function $f \in G_f$ are scalar types[2].

Since we assume that all functions over scalar values are implemented in constant-time, τ takes only constant time on a lifting problem with efficiency guarantee, and thus the synthesized D&C algorithm, which relies on τ to combine results, is efficient. In other words, there is no intermediate data structure produced and consumed during the computation, nor multiple traversals of the same data structure, and thus the computation is efficient according to the idea of fusion and tupling [Pettorossi and Proietti 1996].

Note that the input of p and f are recursive data structures. Therefore, the two functions map a recursive data of any size to a data of constant-size. We call such a function *compressing*, its input size can be much larger than its output size. In this paper we require the output of p and f to be scalar values to simplify theoretical discussion, but our approach can also be generalized to other problems where the output is not a scalar, such as sorting, as long as $p \triangle f$ is compressing.

[2]Please note that products of scalar types are also scalar types.

In the rest of the paper, if not explicitly stated, we assume a lifting problem has efficiency guarantee. Besides, because scalars include tuple of scalars, we further assume that grammars G_f and G_τ are well-structured such that all tuple output are constructed at the top level. Concretely, we assume that both G_f and G_τ are in the form of $S \coloneqq S' \mid S \triangleleft S$ where (1) S is the start non-terminal of the grammar, and (2) the output type of every program in S' is not tuple.

In the following, we shall demonstrate multiple classes of algorithms can be interpreted as D&C and synthesized as a lifting problem by considering different ways of dividing.

Example–Parallelization. We have already seen in Section 2 the example of parallelization of a function p, which is the following lifting problem:

$$\mathsf{LP}(p, \mu) \textbf{ where } \mu((), (ys, zs)) = ys +\!\!+ zs$$

where the first argument of μ, which has type C, is not needed. After f and τ are synthesized, they can be used in the following template to form a parallel program for processing lists.

$$\begin{array}{lcl} dc\ xs@(ys +\!\!+ zs) \mid \mathit{length}\ xs \le k & = & (p\ xs, f\ xs) \\ \mid \textbf{otherwise} & = & \tau((), (dc\ ys, dc\ zs)) \end{array}$$

When the input list is shorter than a threshold k, p and f are directly applied to the list. Since k is a small constant, the input list can be viewed as a scalar and the thus this call is constant-time. When the input list is longer than k, it is divided into two sublists ys and zs, and recursively solved, where the two recursive calls can be executed in parallel. We have also discussed τ is constant time, and thus the algorithm takes $O(n/t)$ on $t = O(n/\log n)$ processors.

Example–Single Pass. Another common used algorithm design paradigm is "single pass", which scans an input list once from the first element to the last element, and iteratively update the result for each element, as shown in the following template.

$$\begin{array}{lcl} sp\ [] & = & (p\ [], f\ []) \\ sp\ (x : xs) & = & \tau(x, sp\ xs) \end{array}$$

Since the list is only scanned once, the time complexity of this algorithm is linear when τ is a constant-time operation. As a result, single-pass is commonly used in designing linear algorithms.

Given a specification program p, we can synthesize a single pass program by solving the following lifting problem.

$$\mathsf{LP}(p, \mu) \textbf{ where } \mu(x, xs) = x : xs$$

In other words, the input list is divided as x and xs which form the original list through the cons operator :. Since x is not a list, it is treated as a complement parameter in C.

Example–Incrementalization. Single pass is a special incremental algorithm, which incrementally update the result when an element is prepended to the list. Interestingly, other incremental algorithms, which are usually not considered as instances of D&C, could also be synthesized via the lifting problem. The basic idea is that we can divide an update data structure as a previous version of the data structure and a modification.

Let C be a domain of updating operations such as "remove the head element" and "change the second element to 3", and $\mu(m, xs)$ apply a modification $m \in C$ to the input data structure xs. Given a specification program p, we can synthesize an incremental algorithm that update the result based on a modification m by solving the following lifting problem.

$$\mathsf{LP}(p, \mu) \textbf{ where } \mu(m, xs) = m\ xs$$

To use the synthesized algorithm, we first apply $p \triangleleft f$ to the initial result on a data structure. Each time the data structure is changed with m, we use $\tau\ m$ to update the result. When the lifting problem has efficiency guarantee, the updating takes constant time.

Example–Segment Trees. A segment tree is a data structure for efficiently answering queries about a specific property of a segement in a possibly long list. For example, we may query "the average value of the segment from the 2nd to the 5000th element in a list". A trivial implementation takes $O(n)$ time where n is the length of the segment. A segment tree reduces the time to $O(\log n)$.

To synthesize the implementation of a segement tree for a specific property of interest, i.e., the average value, most the code is standard, and only two procedures need to be synthesized based on the property. The first procedure calculates the property of a list from those of its sublists. The second update the property of list based on a modification to the list. We can see that the two procedures corresponding to the two examples we have seen before, parallelization and incrementalization. However, we cannot treat the problem as two lifting problems, as the two procedures need to use the same set of auxiliary values. Therefore, we create a new operation that captures the two ways of dividing together. Concretely, given a specification program p that returns a property from a list, the problem of synthesizing a segment tree can be captured as the following lifting problem.

$$
\begin{array}{l}
\mathsf{LP}(p, \mu) \\
\quad \textbf{where} \\
\qquad \mu\ ((s, m), (xs, ys)) \quad \begin{array}{l|lll} s = 1 & = & xs +\!\!+ ys \\ \textbf{otherwise} & = & m\ xs \end{array}
\end{array}
$$

Operation μ takes a selector s as input, behaves as in parallelization when $s = 1$, and behaves as in incrementalization otherwise. In this way, we capture the two operations uniformly as one operation. The synthesized τ could also be used as two different combination operations by passing different values to s and passing random values to unused parameters.

We will see more interesting examples in Section 7.

4 SPECIFICATION WEAKENING: OCCAM-APPROXIMATION THEOREM

Before introducing our approach to the lifting problem in detail, we first describe a theoretical analysis on synthesis using a weaker specification, which is the basis for bounding the error rate of our approach. Readers who are not interested in the theoretical details could safely skip this section and the discussion in error rate in the next section.

4.1 Basics

In general, a program synthesis task can be specified via formula $\exists p \in \mathbb{P}, \varphi(p)$, where program p represents the synthesis target of the task, $\mathbb{P}$ represents the space of considered programs, and $\varphi : \mathbb{P} \mapsto \texttt{Bool}$ is a function defining the validity of programs in $\mathbb{P}$. We say a synthesis task *realizable* if there exists at least a valid program, i.e., formula $\exists p \in \mathbb{P}, \varphi(p)$ is true. In this section, we use Φ to denote the universe of all concerned synthesis tasks. Besides, for simplicity, we assume that there is a generic program space $\mathbb{P}$ for all tasks in Φ and thus represent each synthesis task only by its validation function φ.

A program synthesizer is a function $\mathcal{S} : \Phi \mapsto (\mathbb{P} \cup \{\bot\})$ mapping a synthesis task to either a program or a value $\bot$ to represent synthesis failure. In this paper, we require the *soundness* of the synthesizer, which means the solution returned by a synthesizer is always valid, i.e., $\forall \varphi \in \Phi, \mathcal{S}(\varphi) \neq \bot \rightarrow \varphi(\mathcal{S}(\varphi))$. Besides, we say a synthesizer is *complete* if it never fails on realizable tasks, i.e., $\forall \varphi \in \Phi, (\exists p \in \mathbb{P}, \varphi(p)) \rightarrow \mathcal{S}(\varphi) \neq \bot$.

4.2 Weaker Approximation and Error Rate

As we have seen in the overview, a precise specification may be difficult to be used for synthesis, and we would like to approximate it with a weaker one. We define the method to derive a weaker specification as a weaker approximation.

Definition 4.1 (Weaker Approximation). A weaker approximation $\#(\cdot) : \Phi \to \Phi$ is a function such that $\varphi(p) \Rightarrow \#\varphi(p)$ for any program p.

Though synthesis with a weaker approximation reduces the complexity of the synthesis task, it is possibly unsound since the program synthesized from the weaker task $\#(\cdot)$ may not be a valid solution to the original task. Therefore, to ensure the practical effect, the error rate of this method should be bounded, i.e., it should seldom return an invalid program for the original task.

Furthermore, in this paper we concern only tasks whose smallest solution is under a threshold lim_s. This is because in practice the scalability of any synthesizer is limited, and if the smallest solution is larger than a certain threshold, any synthesizer would fail. In our D&C synthesis process, we always use a precise specification in the last step by filling the previously synthesized programs into the original specification. If any previous step with a weak specification synthesizes an incorrect program, the last step would fail. Therefore, the error rate makes no difference on the tasks whose smallest solutions are large. We use $\mathbb{P}_{\leq k}$ to represent the set of programs no larger than k.

We assume the original synthesis task φ is sampled from some unknown distribution $\tilde{\Phi}$ over Φ. Based on the above discussion, we have the following definition.

Definition 4.2 (Bounded Error Rate). We say the error rate of synthesizer $\mathcal{S}$ under a weaker approximation $\#(\cdot)$ is bounded by ϵ if and only if the following condition holds.

$$\Pr_{\varphi \sim \tilde{\Phi}} \left[\neg\varphi(\mathcal{S}(\#\varphi)) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*) \right] \leq \epsilon \tag{7}$$

The above condition of bounding error rate is usually difficult to be analyzed directly, as it involves φ, which is already too complex to be directly used in synthesis. Furthermore, the behavior of a synthesizer is usually complex and difficult to analyze as well. To bound the error rate, we need to find conditions that do not use φ or $\mathcal{S}$.

4.3 Sufficient Conditions for Bounding the Error Rate

For simplicity, we start with the cases where synthesizer $\mathcal{S}$ is complete. We will extend our results to incomplete synthesizers in Section 4.5. We find sufficient conditions that do not use φ and $\mathcal{S}$ in three steps, which gradually remove φ and $\mathcal{S}$.

Step 1: remove the φ check of the synthesis result.

In the first step, we use an arbitrary valid program p^* to replace a specification φ. When a synthesis result does not satisfy φ, it must be different from p^*, i.e., the probability of being different from p^* is no smaller than the error rate. Based on this idea, we can prove the following lemma, which removes the φ check of the synthesis result.

Lemma 4.3. *The following formula is a sufficient condition of Formula 7.*

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \Pr_{\varphi \sim \tilde{\Phi}} \left[\mathcal{S}(\#\varphi) \neq p^* \mid \varphi(p^*) \right] \leq \epsilon \tag{8}$$

Due to space limit, we move the proofs to lemmas and theorems to Appendix C.

Step 2: remove the invocation to synthesizer $\mathcal{S}$.

To remove the invocation to synthesizer $\mathcal{S}$, we define an easy-to-analyze set that includes all possible synthesis results when a valid program p^* is given, call a *synthesis domain*, and use the synthesis domain to replace the invocation to $\mathcal{S}$.

Definition 4.4 (Synthesis Domain). Given a synthesizer $\mathcal{S}$ and a program p^*, the synthesis domain of $\mathcal{S}$, denoted as $\mathcal{S}^D(p^*)$, is a set satisfying

$$\mathcal{S}^D(p^*) = \{\mathcal{S}(\varphi) \mid \varphi \in \Phi, \varphi(p^*)\}.$$

We notice that for many synthesizers, it is easy to analyze what programs are possibly in the synthesis domain, and this domain is also bounded, as demonstrated by the following example.

Example 4.5. Let $\mathcal{S}_E$ be the basic enumerative synthesizer. Given task φ, $\mathcal{S}_E$ enumerates programs in $\mathbb{P}$ from small to large and returns the first program p satisfying $\varphi(p)$ as the result. Given that p^* is a valid program, the program synthesized by $\mathcal{S}_E$ must be no larger than p^*. Therefore, the size of the synthesis domain $\mathcal{S}_E^D(p^*)$ is bounded via the following inequality.

$$\left|\mathcal{S}_E^D(p^*)\right| \leq \left|\{p \in \mathbb{P} \mid \text{size}(p) \leq \text{size}(p^*)\}\right| \leq 2^{\text{size}(p^*)}$$

where $\text{size}(p)$ is defined as the length of the binary representation of program p.

Using the concept of the synthesis domain, the sufficient condition provided by Lemma 4.3 can be ensured via the following two conditions that do not contain invocations to the synthesizer. Intuitively, a program that does not satisfy the specification would not be returned. If any program in the synthesis domain except p^* has a small probability to satisfy the specification, and the synthesis domain is small (so that the small probabilities do not add up), any program different from p^* is unlikely to be returned.

Lemma 4.6. *For any constants α, β, Formula 7 is satisfied for constant $\epsilon \geq \alpha\beta$ if the following conditions are satisfied.*

- *For any program $p^* \in \mathbb{P}_{\leq lim_s}$, the size of synthesis domain $\mathcal{S}^D(p^*)$ is no larger than α.*
- *For any program $p^* \in \mathbb{P}_{\leq lim_s}$ and any other program p in the synthesis domain, i.e., $p \in \mathcal{S}^D(p^*)/\{p^*\}$, the following formula holds.*

$$\Pr_{\varphi \sim \tilde{\Phi}}[\#\varphi(p) \mid \varphi(p^*)] \leq \beta \tag{9}$$

Step 3: remove the use of φ in the condition.

Formula 9 still uses φ in its condition. To remove this use, we assume some property of the weak approximation $\#(\cdot)$. In our synthesis process, we divide the result program into subparts, and then the precise specification φ for the first subpart can be viewed as a conjunction of two conditions: (1) the solution should satisfy the specification for this subpart (i.e., the weak specification $\#\varphi$), and (2) the solution should make other subparts realizable (called a complementary specification). Since the two conditions are related to different subparts, it is naturally to assume their solution sets are independent. Let $\mathbb{P}_\varphi$ be the solution set of φ, i.e., $\mathbb{P}_\varphi = \{p \mid \varphi(p)\}$. We have the following definition.

Definition 4.7 (Independently Weaker Approximation). We say $\#(\cdot)$ is an independently weaker approximation on distribution $\tilde{\Phi}$ if there exists a complementary specification function $(\cdot)^c : \Phi \mapsto \Phi$ satisfying the following conditions.

- The complementary specification φ^c always provides the constraints ignored by the weaker specification $\#\varphi$, i.e., $\forall \varphi \in \Phi, \forall p \in \mathbb{P}, \varphi(p) = \#\varphi(p) \wedge \varphi^c(p)$
- The solution sets of the weaker specification and the complementary specification are independent under $\tilde{\Phi}$.

$$\forall A, B \subseteq \mathbb{P}, \Pr_{\varphi \sim \tilde{\Phi}}[\mathbb{P}_{\#\varphi} = A] \times \Pr_{\varphi \sim \tilde{\Phi}}[\mathbb{P}_{\varphi^c} = B] = \Pr_{\varphi \sim \tilde{\Phi}}[\mathbb{P}_{\#\varphi} = A \wedge \mathbb{P}_{\varphi^c} = B]$$

Based on the definition, we can prove the following lemma, where $\#\tilde{\Phi}$ is the marginal distribution of $\#\varphi$ defined as $\#\tilde{\Phi}(\varphi') = \Pr_{\varphi \in \tilde{\Phi}}[\varphi' = \#\varphi]$.

Lemma 4.8. *When $\#(\cdot)$ is an independent weaker approximation, the following formula is a sufficient condition of Formula 9.*

$$\Pr_{\varphi' \sim \#\tilde{\Phi}}[\varphi'(p) \mid \varphi'(p^*)] \leq \beta \tag{10}$$

So far, the usages of φ and $\mathcal{S}$ are removed. Analyzing the probability in the sufficient condition provided by 4.8 is easy as it involves only the weaker task $\#\varphi$, whose form is assumed to be simple.

4.4 Limiting the Error Rate via Occam Solver

In the previous section we have proposed two sufficient conditions for bounding the error rate. Now we show the first condition can be easily satisfied when the synthesizer is an *Occam solver* [Ji et al. 2021]. Given a realizable synthesis task, an Occam solver ensures that the synthesis result is at most polynomially larger than the any valid solution.

Definition 4.9 (Occam Solver). For any constant $\alpha \geq 1$ and $c > 0$, a complete synthesizer $\mathcal{S}$ is an α-Occam solver with constant c if the following formula is satisfied[3].

$$\forall \varphi \in \Phi, \forall p^* \in \mathbb{P}, \left(\varphi(p^*) \rightarrow \text{size}(\mathcal{S}(\varphi)) \leq c\,(\text{size}(p^*)^\alpha)\right)$$

By definition, the size of the synthesis result of an Occam solver must be bounded by the size of any valid solution p^*. Therefore, the synthesis domain $\mathcal{S}^D(p^*)$ must also be bounded by the size of p^*, as shown in the following lemma.

Lemma 4.10. *For any α-Occam solver $\mathcal{S}$ with constant c and any program $p^* \in \mathbb{P}$, the following inequalities always hold.*

$$\mathcal{S}^D(p^*) \subseteq \mathbb{P}_{\leq c\text{size}(p^*)^\alpha} \qquad \left|\mathcal{S}^D(p^*)\right| \leq |\mathbb{P}_{\leq c\text{size}(p^*)^\alpha}| \leq 2^{c\text{size}(p^*)^\alpha}$$

Based on the above lemma, we naturally have the following theorem.

Theorem 4.11 (Occam-Approximation Theorem). *The error rate of an α-Occam solver $\mathcal{S}$ with constant c under an independently weaker approximation $\#(\cdot)$ is bounded by $2^{c lim_s^\alpha}\beta$ for constant β, if the following condition holds.*

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \forall p \in \mathcal{S}^D(p^*)/\{p^*\},\ \Pr_{\varphi' \sim \#\tilde{\Phi}}\left[\varphi'(p) \mid \varphi'(p^*)\right] \leq \beta \tag{11}$$

To apply this theorem to estimate the error bound of an Occam solver $\mathcal{S}$ under an independent weaker approximation $\#(\cdot)$, we can proceed in the following two steps.

(1) Construct a proper distribution $\#\tilde{\Phi}$ to model the weaker task met in practice.
(2) Find a small enough constant β satisfying Formula 11, and then get a bound for the error rate by applying the Occam-approximation theorem.

In Section 5.1 and 5.2, we shall show how this theorem guides the design of *AutoLifter*.

4.5 Extensions to Synthesizers that may Failure

In Section 4.3 and 4.4, we derive sufficient conditions for complete synthesizers and Occam solvers. In more general cases, the synthesizer may be incomplete, and an Occam solver may fail to satisfy the size constraint. In this section, we capture these cases with the concept of *probabilistic Occam solver* that bounds the probability of all failures.

Definition 4.12 (Probabilistic Occam Solver). For any constants $\alpha \geq 1, c > 0$, and $0 \leq \gamma \leq 1$, a synthesizer $\mathcal{S}$ is a probabilistic α-Occam solver with constant (c, γ) for distribution $\tilde{\Phi}$ if the following formula is satisfied.

$$\Pr_{\varphi \sim \tilde{\Phi}}\left[\exists p \in \mathbb{P}, \left(\varphi(p) \rightarrow (\mathcal{S}(\varphi) = \bot \vee \text{size}\,(\mathcal{S}(\varphi)) \leq c\text{size}(p)^\alpha)\right)\right] \leq \gamma$$

[3]The original definition of Occam solvers provided by Ji et al. [2021] is for example-based synthesizers with a failure probability. Here we extend it to general synthesizers, and ignore the failure probability for simplicity. The version with a failure probability will be discussed in Section 4.5.

Since the difference between the probabilistic Occam solvers and a standard Occam solver is bounded, the Occam-approximation theorem can be generalized to probabilistic Occam solvers.

Theorem 4.13 (Extended Occam-Approximation Theorem). *The error rate of probabilistic α-Occam solver $\mathcal{S}$ with constant (c, γ) under an independently weaker approximation $\#(\cdot)$ is bounded by $2^{c lim_s^{\alpha}} \beta + \gamma$ with constant β, if the following condition holds.*

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \forall p \in \left(\mathcal{S}^D(p^*) \cup \mathbb{P}_{\leq size(p^*)}\right) / \{p^*\}, \ \Pr_{\varphi' \sim \tilde{\#\Phi}} \left[\varphi'(p) \mid \varphi'(p^*)\right] \leq \beta$$

5 DIVIDE-AND-CONQUER APPROACH TO SYNTHESIS

After explaining our theoretical basis in Section 4, we are ready to introduce the main approach of *AutoLifter*, which is a D&C-style procedure based on two methods *decomposition* and *decoupling*. *AutoLifter* uses these two methods to divide the lifting problem into simpler synthesis subproblems, and their effectiveness is guaranteed by the Occam-approximation theorem proposed in Section 4.4.

We should introduce each method in three steps: (1) how to divide the problem precisely, (2) how to approximate the first subproblem with a weaker approximation, and (3) how to bound the error rate of the first subproblem. For brevity, in this section we use $x, x', x_1, x_2, \ldots$ to represent arbitrary elements in C, $a, a', a_1, a_2, \ldots$ to represent arbitrary elements in A^n, and $\overline{a}, \overline{a'}, \overline{a}_1, \overline{a}_2, \ldots$ to represent arbitrary elements in A^n for some n that should be clear from the context, and omit the respective universal quantifiers.

5.1 Decomposition

Precise subproblems. Recall in our running example, after decomposition, the second specification has a similar, but different form of the original specification. To capture them uniformly, we introduce a generalized version of the lifting problem, which decomposition solves. Concretely, the generalized lifting problem is specified by (1) two functions p, h with the same domain A and (2) an operator μ with some complementary type C and arity n. Its task is to find a lifting function f and an operator τ such that the following formula is satisfied.

$$\varphi_g(f, \tau) : (p \triangle f)\ (\mu\ (x, \overline{a})) = \tau\ \left(x, (h \triangle f)^n\ \overline{a}\right) \tag{12}$$

Clearly, lifting problem $\mathsf{LP}(p, \mu)$ is an instance of this problem, where functions p and h are equal.

Now we discuss how to solve $\varphi_g(f, \tau)$. According to the output type, a valid operator τ must be in the form of $\tau_1 \triangle \tau_2$, where τ_1 and τ_2 calculate the outputs of p and f respectively. *Decomposition* divides the generalized task by first synthesizing τ_1 and a part of the lifting function f which provides auxiliary values for the output of p, and then synthesizing τ_2 with the remaining part of f. The generated subtasks, denoted as $\varphi_{g,1}(f_1, \tau_1)$ and $\varphi_{g,2}[\tau_1, f_1](\tau_2, f_2)$ respectively, is shown below.

$$\begin{aligned} \varphi_{g,1}(f_1, \tau_1) : \quad & \forall x, \forall \overline{a}, \left(p\ (\mu\ (x, \overline{a})) = \tau_1\ \left(x, (h \triangle f_1)^n\ \overline{a}\right)\right) \\ & \wedge \exists f_2 \in G_f, \exists \tau_2 \in G_\tau, \varphi_{g,2}[f_1, \tau_1](f_2, \tau_2) \end{aligned} \tag{13}$$

$$\varphi_{g,2}[f_1, \tau_1](f_2, \tau_2) : \quad \forall x, \forall \overline{a}, (f_1 \triangle f_2)\ (\mu\ (x, \overline{a})) = \tau_2\ \left(x, ((h \triangle f_1) \triangle f_2)^n\ \overline{a}\right) \tag{14}$$

For any valid (f_1, τ_1) to the first task $\varphi_{g,1}$ and any valid (f_2, τ_2) to the corresponding second task $\varphi_{g,2}[f_1, \tau_1]$, *decomposition* constructs (f, τ) for the original task as $(f_1 \triangle f_2, (\tau_1 \circ \sigma_1) \triangle (\tau_2 \circ \sigma_2))$, where functions σ_1, σ_2 simply reorganize the input to match the input types of τ_1 and τ_2[4]. As shown in the following lemma, the above dividing ensures both the soundness and completeness.

[4]Strictly, to ensure the constructed operator τ to be inside grammar G_τ, G_τ is required to include operators for accessing and constructing tuples. Here we implicitly assume the existence of these operators because any non-trivial τ also require them to access the input and construct the output, since both the input and the output of τ are tuples.

Lemma 5.1. *For any generalized lifting problem φ_g that is realizable, decomposition guarantees that (soundness) the constructed program must be valid for φ_g, and (completeness) the first subtask and all possible second subtasks must be realizable.*

Note that after the first subtask is solved, the second task $\varphi_{g,2}[\tau_1, f_1]$ is an instance of the generalized task specified by $f_1, h \triangleleft f_1$, and μ. Therefore, it can be solved recursively.

Weaker approximation. Solving the first subtask $\varphi_{g,1}$ of decoupling is difficult since it involves the existential constraint on f_2 and τ_2. Therefore, *decomposition* introduces a weaker approximation $\#\varphi_{g,1}$ for the first subtask, where the existential constraint is directly ignored.

$$\#\varphi_{g,1}(f_1, \tau_1) : p\ (\mu\ (x, \overline{a})) = \tau_1\ \left(x, (h \triangleleft f_1)^n\ \overline{a}\right) \tag{15}$$

Clearly, a valid (f_1, τ_1) for $\#\varphi_{g,1}$ may not satisfy the existential constraint in subtask $\varphi_{g,1}$ and thus may make the second subtask unrealizable. To avoid this case, we apply the Occam-approximation theorem and show that the error rate is bounded when $\#\varphi_{g,1}$ is solved by an Occam solver.

Bounding the error rate. To solve the decomposed problems, we need a synthesizer for $\#\varphi_{g,1}(f_1, \tau_1)$. In this part we analyze that to bound the error rate, the only requirement over the synthesizer is that it has to be an Occam solver.

For simplicity, we discuss only a special case here where (1) the source domain A is `List`, (2) the output types of p, h are both `Int`, and (3) the output of each function $f \in G_f$ and $\tau \in G_\tau$ is a tuple of integers. By the assumption of grammar G_f and G_τ made in Section 3, the third assumption implies that each f or τ is a tuple of functions with output type `Int`. The discussion below can be naturally generalized to other data types. We discuss only standard Occam solvers first, and later we discuss how to map the requirements to probabilistic Occam solvers.

The Occam-approximation theorem requires a distribution $\#\tilde{\Phi}_{g,1}$ to model the task $\#\varphi_{g,1}$ met in practice. To build this distribution, we assume each related black-box function (including input functions $(p \circ \mu)$ and h, and primitive functions in grammars G_f and G_τ) to be random, i.e., each of them is drawn from a uniform distribution over all functions with the same type. Specially, because $(p \circ \mu)$ is used as a whole in $\#\varphi_{g,1}$, we regard it as a single black-box function for simplicity.

To ease the calculation of the distribution, we assume the domain of lists is limited to be finite by parameter L, representing the size limit for lists. Besides, we use parameter N to denote the number of integers. We also assume the weaker approximation is independent since the precise specification $\varphi_{g,1}$ is a conjunction of two conditions related to two subparts of the result program.

We prove that under model $\tilde{\Phi}_{g,1}$, the error rate of $\#\varphi_{g,1}$ is bounded when an Occam solver is used, as shown in the following theorem. Because both the range of integers and the size of lists can be very large, the bound given by the lemma is small when large enough N and L are used.

Theorem 5.2. *For any scalability threshold lim_s, any limits N and L over the size of integers and lists, and any α-Occam solver $\mathcal{S}$ with constant c, the error rate of $\#\varphi_{g,1}$ and $\mathcal{S}$ under model $\tilde{\Phi}_{g,1}$ is bounded by the following formula.*

$$2^{c lim_s^\alpha} \left(N^{-N+1} + N \exp\left(-N^{L-2}\right)\right)$$

Proof Sketch. To prove Theorem 5.2, we prove that the probability for any program in the synthesis domain to be valid in model $\#\tilde{\Phi}_{g,1}$ is bounded under the condition that (f_1^*, τ_1^*) is valid (i.e., Formula 11) and then apply the Occam-approximation theorem. In our case, the considered probability can be simplified into the following formula, where all related functions are random.

$$\Pr\left[\forall x, \forall \overline{a}, \tau_1\ \left(x, (h \triangleleft f_1)^n\ \overline{a}\right) = \tau_1^*\ \left(x, (h \triangleleft f_1^*)^n\ \overline{a}\right)\right] \leq \beta \tag{16}$$

There are two possible cases. First, when $\tau_1 \neq \tau_1^*$, Formula 16 requires two random functions τ_1 and τ_1^* to output the same on a series of input. Second, when $\tau_1 = \tau_1^*$ but $f_1 \neq f_1^*$, for each input $(x, \overline{a})$,

Formula 16 requires a random function τ_1 always to output the same on two inputs generated by another two random functions $\left(x, (h \triangleq f_1)^n\ \overline{a}\right)$ and $\left(x, (h \triangleq f_1^*)^n\ \overline{a}\right)$. Intuitively, both cases lead to low probability events due to the randomness of functions. □

5.2 Decoupling

Precise subproblems. Rule *decoupling* cares about the weaker task $\#\varphi_{g,1}$ (Formula 15) generated by *decomposition*. It splits this task by first finding a proper lifting function f_1 and then synthesizing a corresponding operator τ_1. The specifications of the two generated subtasks are shown below. We name them as φ_f and φ_τ since they are for the lifting function f and the operator τ respectively.

$$\varphi_f(f_1): \quad \exists \tau_1 \in G_c, p\ (\mu\ (x, \overline{a})) = \tau_1\ \left(x, (h \triangleq f_1)^n\ \overline{a}\right) \tag{17}$$

$$\varphi_\tau[f_1](\tau_1): \quad p\ (\mu\ (x, \overline{a})) = \tau_1\ \left(x, (h \triangleq f_1)^n\ \overline{a}\right) \tag{18}$$

For any valid f_1 for the first subtask and any valid τ_1 for the corresponding second subtask, *decoupling* returns (f_1, τ_1) as the result for the original task $\#\varphi_{g,1}$. It is straightforward to verify the soundness and the completeness of this split, as shown in the following formula.

Lemma 5.3. *For any synthesis task in the form of $\#\varphi_{g,1}$ that is realizable, decoupling guarantees that (soundness) the constructed program must be valid for $\#\varphi_{g,1}$, and (completeness) the first subtask and all possible second subtask must be realizable.*

Weaker approximation. Solving subtask φ_f is still difficult due to the existential constraint on τ_1. *Decoupling* introduces a weaker approximation $\#\varphi_f$ utilizing the fact that τ_1 is a function, which requires the output of τ_1 (i.e., $p\ (\mu\ (x, \overline{a}))$) to be equal to the same input (i.e., $(x, (h \triangleq f_1)^n\ \overline{a})$).

$$\#\varphi_f(f_1): (h \triangleq f_1)^n\ \overline{a} = (h \triangleq f_1)^n\ \overline{a'} \rightarrow p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'})) \tag{19}$$

Note that $\#\varphi_f$ only ensures the existence τ_1 among all possible functions, and such a function may not exist in grammar G_τ. Therefore, a valid f_1 for $\#\varphi_f$ may not satisfy the existential constraint in φ_f. Similar to *decomposition*, we apply the Occam-approximation theorem and show that the error rate of this approximation is bounded when a proper synthesizer is used.

Bounding the error rate. In this part we show that to bound the error rate, we need the synthesizer for $\#\varphi_f$ to satisfy two conditions: (1) the synthesizer is an Occam solver, and (2) the synthesizer is minimal, as defined later.

We start with a property of f to simplify discussion. By the assumption on G_f made in Section 3, each function f in G_f is in the form of a tuple of unit functions. We use $U(f)$ to denote the set of units in function f. For any two lifting functions f, f' formed by the same set of units, i.e., $U(f) = U(f')$, because they provide the same the auxiliary values, they must be both valid or both invalid for tasks φ_f and $\#\varphi_f$, i.e., $\varphi_f(f) \Leftrightarrow \varphi_f(f') \wedge \varphi_f(f) \Leftrightarrow \varphi_f(f')$. Therefore, in the discussion below, we simply regard two lifting functions formed by the same set of units as the same.

To conduct the analysis, similar to the analysis on *decomposition*, we (1) assume the weaker approximation given by decoupling is independently weaker, (2) assume all domains are limited by N (the number of integers) and L (the size limit of lists), and (3) model marginal distribution $\#\tilde{\Phi}_f$ by assuming each black-box function to be drawn from a uniform distribution over functions with the same type. We prove that under model $\tilde{\Phi}_f$, the error rate of approximation $\#\varphi_f$ is bounded when the used synthesizer is not only an Occam solver but also a *minimal* synthesizer. We say a synthesizer *minimal* if it always synthesizes a lifting function formed by a minimal set of units.

Definition 5.4. Synthesizer $\mathcal{S}$ is *minimal* if for any weaker task $\#\varphi_f$ (Formula 19) generated by *decoupling*, condition $\forall f \in G_f, \left(\#\varphi_f(f) \rightarrow U(f) \not\subset U(\mathcal{S}(\#\varphi_f))\right)$ is always satisfied.

The following theorem provides an upper-bound for the error rate in this case. Similar to the analysis on *decomposition*, this bound is small when large enough N and L are used.

Theorem 5.5. *For any scalability threshold lim_s, any limits N, L, any α-Occam solver $\mathcal{S}$ with constant c that is also minimal, the error rate of $\#\varphi_f$ and $\mathcal{S}$ under model $\tilde{\Phi}_f$ is bounded by the following formula.*

$$2^{c\,lim_s^{\alpha}} \times 2\left(N^{-N} + N^{s}\exp\left(-N^{L-s}\right)\right), \textit{where } s = 1 + lim_s + c\,lim_s^{\alpha}$$

Proof Sketch. Similar to *decomposition*, to prove Theorem 5.5, we prove that the probability for any program in the synthesis domain to be valid in model $\#\tilde{\Phi}_f$ is bounded under the condition that some other program f_1^* is valid (i.e., Formula 11) and then apply the Occam-approximation theorem. In this case, the concerned probability is equal to $\Pr[\#\varphi_f(f_1) \mid \#\varphi_f(f_1^*)]$.

There are also two possible cases. First, when $U(f_1^*) \subset U(f_1)$, f_1 must be valid for $\#\varphi_f$ since it provides strictly more auxiliary values than f_1^*, a lifting function known to be valid. Therefore, $\Pr[\#\varphi_f(f_1) \mid \#\varphi_f(f_1^*)]$ is equal to 1. Fortunately, because the synthesizer $\mathcal{S}$ is required to be minimal, f_1 cannot be in synthesis domain $\mathcal{S}^D(f_1^*)$ when $U(f_1^*) \subset U(f_1)$, and thus this case can be ignored.

Second, when $U(f_1^*) \not\subset U(f_1)$, the derivation is much more complex. For simplicity, we illustrate the main insight on a simpler probability $\Pr[\#\varphi(f_1)]$ where the condition is directly ignored. This probability can be unfolded and transformed into the following form.

$$\Pr\left[\forall x \in C, \forall \overline{a}, \overline{a'} \in A^n, \left(h^n\ \overline{a} = h^n\ \overline{a'} \wedge p\ (\mu\ (x, \overline{a})) \neq p\ (\mu\ (x, \overline{a'})) \rightarrow {f_1}^n\ \overline{a} \neq {f_1}^n\ \overline{a'}\right)\right]$$

In this probability, ${f_1}^n$ is required to output differently on all pairs of $(\overline{a}, \overline{a'})$ satisfying some premise.

- On the one hand, this probability is positively related to the size of the output domain of f_1, since the larger the output domain is, the more likely ${f_1}^n$ outputs differently will be.
- On the other hand, this probability is negatively related to the size of the input domain of f_1, since the larger the input domain is, the more pairs of considered $(\overline{a}, \overline{a'})$ will be, and thus the more strict the constraint on f_1 will be.

Here we utilize the *compressing* property mentioned in Section 3 of function f_1. In model $\tilde{\Phi}_f$, the input of f_1 is a list, whose number is at least N^L, and the output of f_1 is a constant number of integers, whose number is N^t for some constant t. Therefore, when L is large, the input domain of f_1 is much larger than the output domain, which makes $\#\varphi_f(f_1)$ a low probability event. □

The compressing property used in the above proof is crucial to the effect of *decoupling*. Without this property, we can find that any injective function (e.g., the identity function $id\ x \coloneqq x$) must be valid for the weaker task $\#\varphi_f$, and clearly, most of them will be invalid for the first subtask φ_f when the expressiveness of candidate operators in G_τ is limited.

5.3 Synthesis Framework of *AutoLifter*

AutoLifter repeatedly uses *decomposition* and *decoupling* to decompose the lifting problem into simpler subtasks, as shown in Algorithm 1. We assume the error rates of both rules are bounded and thus directly ignore those cases where an invalid program is synthesized from the weaker task.

In Algorithm 1, function `Decomposition` (Lines 6-10) applies *decomposition* to solve the generalized lifting problem φ_g, and function `Decoupling` (Lines 1-5) applies *decoupling* to solve the weaker task $\#\varphi_{g,1}$ generated by *decomposition*. Specially, when the lifting function f_1 is not necessary (i.e., *null*), both functions will directly return a trivial solution to avoid extra invocations (Lines 3, 8). Besides, Algorithm 1 is configured by two client synthesizers $\mathcal{S}_f$ and $\mathcal{S}_\tau$, which are used for the weaker task $\#\varphi_f$ and the second subtask φ_τ generated by *decoupling* (Lines 2, 5).

Algorithm 1: The synthesis framework of *AutoLifter*.

Input: A lifting problem $\mathsf{LP}(p, \mu)$ and two synthesizers $\mathcal{S}_f$ and $\mathcal{S}_\tau$ for subtasks generated by *decoupling*.
Output: A solution (f, τ) to lifting problem $\mathsf{LP}(p, \mu)$.

```
Function Decoupling(p, h, μ):                                  // Apply decoupling.
    #φ_f ← the weaker task (Formula 19) generated by decoupling;
    if null is valid for #φ_f then f_1 ← null else f_1 ← S_f(#φ_f);
    φ_τ ← the second subtask generated by decoupling when f_1 is given;
    return (f, S_τ(φ_τ));
Function Decomposition(p, h, μ):                               // Apply decomposition.
    (f_1, τ_1) ← Decoupling(p, h, μ);
    if f_1 = null then return (null, τ_1 △ null);
    (f_2, τ_2) ← Decomposition(f_1, h △ f_1, μ);
    return (f_1 △ f_2, (τ_1 ∘ σ_1) △ (τ_2 ∘ σ_2)), where σ_1, σ_2 are used to reorganize the input type;
return Decomposition (p, p, μ);
```

Requirements on client synthesizers. We apply Theorem 5.2 and 5.5 to Algorithm 1 to bound the error rate of each usage of *decomposition* and *decoupling*. For *decoupling* in function `Decoupling`, Theorem 5.5 requires client synthesizer $\mathcal{S}_f$ to be an Occam solver and a minimal synthesizer.

For *decomposition* in function `Decomposition`, Theorem 5.2 requires function `Decoupling` to be an Occam solver. We relax this requirement to probabilistic Occam solvers by the extended Occam-approximation theorem (Theorem 4.13) and then prove that `Decoupling` is a probabilistic Occam solver when (1) the error rate of *decoupling* is bounded by the Occam-approximation theorem, and (2) client synthesizer $\mathcal{S}_\tau$ is a probabilistic Occam solver.

Intuitively, for any task to `Decoupling` with the smallest solution (f_1^*, τ_1^*), the first condition ensures that the synthesized f_1 is equal to f_1^* with a high probability, and the second condition ensures the synthesizer τ_1 to be at most polynomially larger than τ_1^* when $f_1 = f_1^*$. Therefore, the synthesized (f_1, f_1^*) is at most polynomially larger than (f_1^*, τ_1^*) with a high probability, which implies that `Decoupling` is a probabilistic Occam solver. More details can be found in Appendix.

5.4 Synthesizers

As discussed above, *AutoLifter* requires two client synthesizers $\mathcal{S}_f$ and $\mathcal{S}_\tau$ to solve the weaker task $\#\varphi_f$ and the second subtask $\varphi_\tau[f_1]$ generated by *decoupling*. $\mathcal{S}_f$ has to be an Occam solver and a minimal synthesizer. $\mathcal{S}_\tau$ has to be a probabilistic Occam solver.

We use the CEGIS framework [Solar-Lezama et al. 2006] to convert both synthesis tasks into example-based tasks, i.e., synthesizing a program to satisfy a set of given examples. For the second subtask $\varphi_\tau[f_1]$ (Formula 18), an example $(x, \overline{a})$ requires $p\ (\mu\ (x, \overline{a})) = \tau_1\ (x, (h \triangle f_1)^n\ \overline{a})$, which can be regarded as input-output example $(\mathsf{F}(h \triangle f)\ e) \mapsto p\ (m\ e)$. Therefore, we use *PolyGen* [Ji et al. 2021], a state-of-the-art probabilistic Occam solver based on input-output examples, as $\mathcal{S}_\tau$.

For the weaker subtask $\#\varphi_f$ (Formula 19), an example $(x, \overline{a}, \overline{a'})$ requires $f_1{}^n\ \overline{a} \neq f_1{}^n\ \overline{a'}$. Because there is no specialized synthesizer for examples in this form, we implement $\mathcal{S}_f$ based on a state-of-the-art general-purpose synthesizer *observational equivalence (OE)* [Udupa et al. 2013]. *OE* enumerates programs by combining enumerated programs with language constructs and prunes off those duplicated programs that output the same on all given examples. Because *OE* enumerates programs from small to large, it is a 1-Occam solver and a minimal synthesizer.

As mentioned in Section 2.5, we integrate a specialized pruning strategy *observational covering* into *OE* to further improve its efficiency. *Observational covering* utilizes the structure of the lifting function f, which is formed by a tuple of functions, and thus prunes off some non-optimal tuples

while forming f. Since the description of observational covering in Section 2.5 also applies to the general case of the lifting problem, we do not explain its detail again.

5.5 Discussion on the Usage of the Occam-Approximation Theorem

The Occam-approximation theorem serves as a possible guidance to the design of program synthesizers in a top-down manner. For a synthesis task where the scale of the result program may be large, we can divide the result program into a sequence of subparts, and guided by the Occam-approximation theorem, derive weaker and simpler specification and design synthesizers for the subparts except for the last one, and the bounded error rate ensures the synthesizer would not fail frequently. Similarly, for a synthesis task where the specification is too complex, we can also synthesize with a weaker specification, and the Occam-approximation theorem helps bound the probability of returning an incorrect program.

Our theorem requires a distribution modeling for the practical synthesis tasks, and in this paper, we consider a simple model which assumes the complete randomness of semantics. The shortage of this model is that it does not consider domain knowledge and thus may be inaccurate in specific domains. For example, in our dataset, integer arithmetic operators are always available in both G_f and G_τ. At this time, when lifting function f is a valid program for the weaker task $\#\varphi_f$ generated by *decoupling* (Formula 19), another lifting function f' defined as $f'\ x = f\ x + 1$ must also be valid. Such a relation led by the semantics of + violates the randomness in our model. Fortunately, this flaw does not matter since the difference between f and f' can be corrected by arithmetic operators in G_τ. Even though f' is synthesized instead of f, its output w can still be used as the corresponding output of f as $w - 1$. Therefore, this case will not lead to a fault, and as shown in Section 7, *AutoLifter* never synthesizes an invalid program from the weaker approximation in our evaluation. Designing a more precise model incorporating domain knowledge is future work.

6 IMPLEMENTATION

Our implementation of *AutoLifter* focuses on homomorphisms from lists to tuples of integers. It can be generalized to other cases if the corresponding types, operators, and grammars are provided.

Verification. To use the CEGIS framework, a verifier is required to verify whether the candidate program synthesized from the examples is valid for the original synthesis task. *AutoLifter* can be combined with any off-the-shelf verifiers. For example, when the given function p and operator μ can be symbolically executed, the verifier for both tasks $\#\varphi_f$ and φ_τ can be implemented via bounded model checking [Biere et al. 2003]. However, because both p and μ are treated as black-box by all the other components in *AutoLifter*, we also provide a probabilistic verifier under the black-box setting to keep the generality of *AutoLifter*.

Our verifier is based on the correctness guarantee provided by Occam solvers. Ji et al. [2021] proves that the error rate of the program synthesized from random examples is bounded on the original task when (1) the number of examples is larger than a threshold, and (2) the example-based synthesizer is an Occam solver. To get this guarantee for the synthesis result, we need only to ensure each candidate program is verified on enough random examples, since the example-based synthesizers for both tasks $\#\varphi_f$ and φ_τ are already Occam solvers. Our verifier determines a proper number of examples by iterations, and more details on it can be found in Appendix B.1.

Grammars. A lifting problem requires two grammars G_f and G_τ to specify the spaces of candidate lifting functions and operators. We take the grammar used by *DeepCoder* [Balog et al. 2017] as G_f. This grammar contains 17 list-related operators, including common higher-order functions (e.g., *map* and *filter*) and operators performing branching and looping internally (e.g., *count* and *sort*).

We take the grammar for conditional integer arithmetic in SyGuS-Comp [Alur et al. 2019] as G_τ. This grammar contains basic arithmetic operators (e.g., $+, -, \times$, *div*), comparison operators (e.g., $<, \leq$), Boolean operators (e.g., *and*, *or*), and the branch operator *if-then-else*. Therefore, programs in G_τ can express complex programs via nested branch operators.

More details on these two grammars used in *AutoLifter* can be found in Appendix B.2.

7 EVALUATION

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

- **RQ1**: How effective does *AutoLifter* solve lifting problems?
- **RQ2**: How does *AutoLifter* compare against existing synthesizers for automatic parallelization?

7.1 Experimental Setup

Baseline Solvers. We compare *AutoLifter* with two general proposed synthesizers, *Enum* [Alur et al. 2013] and *Relish* [Wang et al. 2018], that can be applied to the lifting problem.

- *Enum* [Alur et al. 2013] is an enumerative solver. Given a lifting problem, *Enum* enumerates all candidate solutions (f, τ) in the increasing order of the size until a valid one is found.
- *Relish* [Wang et al. 2018] is a state-of-the-art synthesizer for relational specifications. It first excludes many invalid programs via a data structure namely *hierarchical finite tree automata* and then searches for a valid program among the automata.

We re-implement both of *Enum* and *Relish* to support the list-related operations used in our paper. Besides, considering the complexity of the full-functional *Relish*, we instantiate it only on the subtask for synthesizing the lifting function, i.e., Formula 19 generated by *decoupling*, because of its simple form. We complete the instantiation of *Relish* into a synthesizer for the lifting problem by combining it with other components in *AutoLifter*, including the synthesis framework comprised by the two rules[5] (i.e., Algorithm 1) and *PolyGen* for synthesizing the operator τ. Note that such a comparison favors *Relish* since its synthesis tasks have been significantly simplified.

We also compare *AutoLifter* with a state-of-the-art synthesizer for automatic parallelization, namely *Parsynt* [Farzan and Nicolet 2017, 2021b]. *Parsynt* is a white-box synthesizer that requires the original program p to be given as single-pass. It uses pre-defined rules to transform the loop body of p, extracts lifting function f directly, and then synthesizes a corresponding operator τ via an inductive synthesizer. There are two versions of *Parsynt* available, where different transformation systems are used. We denote them as *Parsynt17* [Farzan and Nicolet 2017] and *Parsynt21* [Farzan and Nicolet 2021b] respectively, and consider both of them in our evaluation.

Dataset. Our evaluation is conducted on a dataset $\mathcal{D}$ of 57 lifting problems extracted from programming tasks related to three algorithmic problems and five classes of algorithms. Due to the space limit, we introduce these problems in brief. More details can be found in Appendix D.1.

Parallelization is the parallelization problem introduced in Setion 2 and Section 3. We collect 35 programming tasks for parallelization from the datasets used by previous studies [Bird 1989a; Farzan and Nicolet 2017, 2021b][6], including all tasks used by Bird [1989a]; Farzan and Nicolet [2017] and 12 out of 22 tasks used by Farzan and Nicolet [2021b]. The other 10 tasks used by Farzan and Nicolet [2021b] require a more general class of parallel algorithms where the divide operator is

[5] *Relish* guarantees the synthesis result to be the smallest valid program. Therefore, *Relish* is a 1-Occam solver, and thus the effect of the hybrid system is also guaranteed via the Occam-approximation theorem.

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

not determined. Therefore, the application of this algorithm class cannot be expressed by lifting problems and are out of the scope of *AutoLifter*.

Longest Segment Problem is described by a predicate b on lists. Given a list, this problem asks for the length of the longest segment in the list satisfying predicate b. Zantema [1992] proposes three classes of greedy algorithms for predicate b under different conditions. To create synthesizers for the three greedy algorithms, the key is to find three efficient incremental updating procedures based on b, respectively. The synthesis tasks for the three procedures can be reduced to lifting problems. We collect 11 tasks for the longest segment problems from the sample tasks used by Zantema [1992], including 3, 1, and 4 tasks for the three classes of greedy algorithms respectively.

Range Update and Range Query [Bentley 1977] (abbreviated as RANGE) is a classical data structure problem. Given an initial list x, a query function h, and an update function u, the task of RANGE is to process a series of operations in order. There are two types of operations.

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

The RANGE task can be solved by the segment tree introduced in Section 3. Because no previous work on RANGE and segment trees provides a dataset, we search on *Codeforces*[7], a website for competitive programming, using keywords "segment tree" and "lazy propagation"[8], and collect 13 programming tasks for the RANGE problem.

7.2 RQ1: Comparison of Synthesizers for Lifting Problems

Procedure. We compare *AutoLifter* with *Enum* and *Relish* on all benchmarks in $\mathcal{D}$ with a time limit of 300 seconds and a memory limit of 8 GB. To measure the efficiency of the solvers, we record the time cost for each successful synthesis. To reflect the quality of the result, we measure the number of auxiliary values used in each success synthesis, which is the number of integers returned by the lifting functions. We care about this value because the fewer the auxiliary values are, the fewer the values produced and stored will be, and the more efficient the resulting algorithm is likely to be.

Besides, as discussed in Section 6, the default grammar G_f and G_τ used by *AutoLifter* comes from *DeepCoder* [Balog et al. 2017] and SyGuS-Comp [Alur et al. 2019]. Though these two grammars are generally effective, they are not expressive enough for 9 tasks in $\mathcal{D}$ where specialized operators such as regex matching on an integer list are required. Note that *AutoLifter* can be easily extended to these specialized domains by including more operators in the grammar. To evaluate the performance of *AutoLifter* on these tasks, we set up an enhanced setting where missing operators are manually provided to the grammars. Details on these operators can be found in Appendix D.2.

Results. The results of this experiment are summarized as the upper part of Table 1. We manually verify the synthesis results of all synthesizers and confirm that all of them are **completely correct**. Besides, we also manually verify all usages of *decomposition* and *decoupling* in both *AutoLifter* and the hybrid synthesizer *Relish*. We find that the program synthesized from the weaker approximation never goes wrong in this experiment. This result matches our theoretical analysis based on the *AutoLifter* and demonstrates the error rate of *AutoLifter* does not matter in practice.

On the efficiency, *AutoLifter* significantly outperforms all baselines. It not only solves much more benchmarks but also solves faster on those jointly solved ones. Note that on many tasks, *Relish* fails even in synthesizing a part of the lifting function (i.e., task $\#\varphi_f$). This result demonstrates that its scalability is far from enough for solving the lifting problem.

[7] https://codeforces.com/
[8] A common alias of segment trees while solving RANGE tasks.

Table 1. The results of the evaluation.

Solver	Dataset		#Solved		Average Time Cost (s)[1]		Average #Auxiliary Values[1]	
	ID	Size	Base[2]	*Auto*[2]	Base[2]	*Auto*[2]	Base[2]	*Auto*[2]
Exp1 (Section 7.2)[3]								
AutoLifter	$\mathcal{D}$	57	48 (56)		7.40 (8.38)		2.13 (2.30)	
Enum			9 (10)	**48 (56)**	34.3 (33.1)	**0.12 (0.32)**	0.56 (0.60)	0.56 (0.60)
Relish			26 (28)		8.34 (8.12)	**5.91 (5.63)**	1.23 (1.29)	1.23 (1.29)
Exp2 (Section 7.3)[3]								
Parsynt17	$\mathcal{D}_P^-$	20	19	19 (**20**)	15.6 (19.2)	**3.85 (3.76)**	0.94|0.61 (0.89|0.63)[4]	**1.50 (1.47)**
Parsynt21	$\mathcal{D}_P$	35	24	**27 (34)**	5.62 (6.74)	**1.19 (5.46)**	1.21|1.58 (1.25|1.79)[4]	**1.79 (2.21)**

[1] The average includes only the results on the tasks solved by both solvers.
[2] *Base* and *Auto* show the performance of *Baseline* and *AutoLifter* on the same set of tasks, respectively.
[3] For results listed as a (b), a and b represents results under the default and enhanced setting respectively.
[4] The number of auxiliary values used by *ParSynt* is listed as $c|d$, where c and d are the number of auxiliary values provided by the single-pass program and the number of synthesized auxiliary values respectively.

On the quality of the synthesis result, *AutoLifter* always uses the same number of auxiliary values as baseline solvers on those jointly benchmarks, where the program synthesized by *Enum* is guaranteed to be minimal. This result shows that *AutoLifter* is able to find high-quality results.

Under the enhanced setting, the performance of *AutoLifter* is improved. Such a result shows that *AutoLifter* can be improved if missing operators can be automatically inferred. To achieve this, one possible way is to extract useful operators from the user-provided implementation. This will be future work. The only failed task is *longest_odd_(0+1)* constructed by Farzan and Nicolet [2021b], where a correct lifting function is found but *PolyGen* fails in finding the corresponding operator τ.

7.3 RQ2: Comparison with Synthesizers for Automatic Parallelization

Procedure. We compare *AutoLifter* with the two versions of *Parsynt* on those tasks for parallelization in our dataset. We denote this subset as $\mathcal{D}_P$. Because *ParSynt* requires the original program p to be single-pass, we provide such an implementation for each task in $\mathcal{D}_P$ to invoke *ParSynt*[9]. Note that such a comparison favors *ParSynt* because some required auxiliary values are directly provided to *ParSynt* in the single-pass implementation.

We failed in installing *Parsynt17* because of some issues on the dependencies. The authors of *Parsynt17* confirmed but have not solved this problem. So we compare *AutoLifter* with *Parsynt17* only on its original dataset $\mathcal{D}_P^-$ 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. On efficiency, though about 60% and 40% auxiliary values as well as the syntax information are directly provided to *Parsynt17* and *Parsynt21* respectively, *AutoLifter* still solves an equal or higher number of tasks and achieves a much faster speed on those jointly solved tasks.

On the quality of the synthesis result, *AutoLifter* uses fewer auxiliary values than both versions of *Parsynt*. This is because the syntax information may misleed *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. It can be specified by single-pass program $(ls \triangleleft max)\ l = fold\ (\oplus)\ (false, -\infty)\ l$, 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 element, *Parsynt* will extract *last* l as an auxiliary

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

value. However, this value is unnecessary because $ls\ (l_1 +\!\!+ l_2)$ is always equal to $(ls\ l_2) \wedge (max\ l_1 \leq max\ l_2)$. *AutoLifter* can find this simpler solution as it synthesizes directly from the semantics.

7.4 Case Study

We make a case study on two tasks in our dataset, which shows that *AutoLifter* (1) can find results that are counter-intuitive on syntax, and (2) can solve problems that are hard even for world-level players in competitive programming.

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

It is not easy to calculate the maximum segment product in parallel. According to the experience on solving *mss*, one may choose the maximum prefix/suffix product as the auxiliary values. However, these two functions are not enough. The tricky point here is that, the maximum segment product is also related to the **minimum** prefix/suffix product, which is counter-intuitive. This is because both the minimum suffix product and the maximum prefix product can be negative integers with a large absolute value. Their product will flip back the sign and resulting in a large positive number.

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

$$\begin{aligned} f\ xs = \quad & (maximum\ (scanl\ (\times)\ l), maximum\ (scanl\ (\times)\ l), minimum\ (scanl\ (\times)\ l), \\ & minimum\ (scanl\ (\times)\ l), head\ (scanr\ (\times)\ l)) \end{aligned}$$

Longest Segment Problem 22-2. This second problem is proposed by Zantema [1992], which is used as the second example on Page 22 of the paper. The task is to find a linear-time algorithm for the length of the longest segment s satisfying $minimum\ s + maximum\ s > length\ s$ 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. The result is that only 26 out of 243 teams successfully solves this problem within 5 hours.

The third class of algorithms proposed by Zantema [1992] can be used to solve this problem. At this time, the task becomes to find function f and a constant-time operator τ such that for any lists xs, ys and integer x satisfying $x < minimum\ xs \wedge x \leq minimum\ ys$, the formula below is satisfied.

$$(p\ \triangle\ f)\ (xs +\!\!+ [x] +\!\!+ ys) = \tau\ \Big(x, \big((p\ \triangle\ f)\ xs, (p\ \triangle\ f)\ ys\big)\Big)$$

where p represents any correct program for this longest segment problem. However, even though the problem is transformed by the algorithm class, finding proper f and τ is still difficult. We encourage the readers to try this task before moving to the discussion below.

AutoLifter can find lifting program $f\ xs := (length\ xs, maximum\ xs)$ and a correct operator τ using only 14.33 seconds. The structure of τ is complex, and here we only explain its component for calculating *lsp*. When $maximum\ xs \geq maximum\ ys$, τ deals with the following 2 cases:

- When $x + maximum\ xs > length\ xs + 1$, τ returns the following value as the result.

  $$\tau_1\ (x, ((lsp_L, (len_L, max_L)), (lsp_R, (len_R, max_R)))) = \\ max\ lsp_R\ (min\ (len_L + len_R + 1)\ (x + max_L - 1))$$

  There are only three possible cases for the longest valid segment: the longest valid segment s_L in xs, the longest valid segment s_R in ys, and the longest valid segment s_x contains element x. First, since $x + max_L > len_L + 1$, segment $xs +\!\!+ [x]$ is valid and thus s_L is no

longer than s_x. Second, we can prove that s_x must be the prefix of the whole list with length $\min(len_L + len_R + 1, x + max_L - 1)$, as this list has already included the largest element max_L in the whole list. Therefore, the result is the longer one between s_R and s_x.

- Otherwise, τ returns $\max(lsp_L, lsp_R)$ as the answer. This is correct because at this time, any valid segment s containing x must be no longer than len_L. For any such segment s, let s' be any segment in xs that contains the largest element in xs and has the same length with s. s' must also be valid, as $length\ s' = length\ s$, $minimum\ s' \geq minimum\ s$ and $maximum\ s' \geq maximum\ s$. Therefore, the result must be $\max(lsp_L, lsp_R)$ at this time.

For the case where $maximum\ xs < maximum\ ys$, τ will deal with it symmetrically. As we can see from this analysis, the correct τ for this problem utilizes several tricky properties, and finding such an operator is hard for a human user. In contrast, *AutoLifter* is able to solve this problem quickly.

8 RELATED WORK

D&C Synthesis. In this paper, we consider a broad class of D&C algorithms, and thus our paper is related to previous studies on synthesizing algorithms related to D&C. First, several approaches [Farzan and Nicolet 2017; Fedyukovich et al. 2017; Morita et al. 2007; Raychev et al. 2015] have been proposed to synthesize parallel algorithms in the form of list homomorphism, which is a subclass of D&C algorithms as discussed in Section 2. All of these approaches are white-box and require the input program to be implemented as single-pass. Compared with them, *AutoLifter* is the first black-box synthesizer that does not require single-pass implementations, and as shown in our evaluation, *AutoLifter* achieves competitive performance with state-of-the-art white-box synthesizers for parallelization.

Second, there exist multiple synthesizers for parallelization and other subclasses of the D&C algorithms that do not support the synthesis of the lifting function, including parallelization [Ahmad and Cheung 2018; Radoi et al. 2014; Smith and Albarghouthi 2016], structural recursion [Farzan et al. 2022; Farzan and Nicolet 2021a], and incrementalization [Acar et al. 2005; Liu and Stoller 2003]. These approaches will fail when the output of the input program cannot be directly calculated in a D&C manner. Compared with these approaches, *AutoLifter* not only supports a larger class of algorithms but can also automatically find proper auxiliary values when a lifting function is necessary.

AutoLifter assumes the divide operator is given (i.e., operator μ in the lifting problem) while synthesizing D&C algorithms. In this sense, there are two approaches that support a more general task where the divide operator is to be synthesized as well [Farzan and Nicolet 2021b; Miltner et al. 2022]. The two approaches and our approach are complementary because the approach by Farzan and Nicolet [2021b] requires a single-pass implementation and the approach by Miltner et al. [2022] does not support to synthesize the lifting function. A possible future direction is to combine these approaches with *AutoLifter*.

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

Compared with *AutoLifter*, these approaches can achieve more refined guarantees on the efficiency via type systems. However, these approaches need to synthesize the whole program from the start, where scalability becomes an issue. As far as we are aware, so far none of these approaches could scale up to synthesizing efficient D&C algorithms as our approach does.

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

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

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

CVC5 [Barbosa et al. 2022] is another general-purposed synthesizer based on SMT solvers. It cannot be applied to our task because the lifting problem involves complex operations on recursive data structures that are difficult to model in SMT-Lib. Besides, there are also solvers for synthesizing programs related to recursive data structures, including *DeepCoder* [Balog et al. 2017], *Myth* [Osera and Zdancewic 2015], λ^2 [Feser et al. 2015], *Refazer* [Rolim et al. 2017], and *Burst* [Miltner et al. 2022]. All of these solvers are based on input-output examples, which are unavailable in lifting problems and the subtask $\#\varphi_f$ for the lifting functions generated by *decoupling*.

9 CONCLUSION

In this paper, we study the problem of synthesizing efficient algorithms on data structures. Motivated by the observation that the idea of D&C generally exists in a board class of efficient algorithms, we capture the usage of D&C as the lifting problem and propose a novel synthesizer *AutoLifter* for it. To solve the lifting problems efficiently, we use two techniques, decomposition and decoupling, to divide difficult tasks into simpler subtasks and replace complex subtasks with weaker approximations. Besides, we also prove the Occam-approximation theorem to bound the error rate of *AutoLifter* while using weaker approximations and propose a pruning strategy namely observational covering for implementing efficient synthesizers for the generated subtasks. Our evaluation shows that *AutoLifter* can efficiently synthesize algorithms for various algorithm classes.

Though this paper focuses on lifting problems, many techniques proposed in *AutoLifter* are general and can be potentially applied to other tasks. For example, the idea of weaker approximation and the Occam-approximation theorem can be used to simplify other complex tasks, and the technique of decoupling can separate the composition of two unknown functions in other relational synthesis tasks when the compressing property holds. Exploring these applications is future work.

The source code of our implementation, all experimental data, and the appendix paper can be found at Anonymous [2022].

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

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Commun. ACM* 61, 12 (2018), 84–93. https://doi.org/10.1145/3208071

Anonymous. 2022. Supplementary Material. https://github.com/autolifter/AutoLifter.

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

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

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

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

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

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

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 244–259. https://doi.org/10.1145/3519939.3523726

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 540–555. https://doi.org/10.1145/3062341.3062355

Azadeh Farzan and Victor Nicolet. 2021a. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 832–855. https://doi.org/10.1007/978-3-030-81685-8_39

Azadeh Farzan and Victor Nicolet. 2021b. Phased synthesis of divide and conquer programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 974–986. https://doi.org/10.1145/3453483.3454089

Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 572–585. https://doi.org/10.1145/3062341.3062382

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

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. Generalizable synthesis through unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485544

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 253–268. https://doi.org/10.1145/3314221.3314602

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

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

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

Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 146–155. https://doi.org/10.1145/1250734.1250752

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Alberto Pettorossi and Maurizio Proietti. 1996. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Comput. Surv.* 28, 2 (1996), 360–414. https://doi.org/10.1145/234528.234529

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

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

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: OBSERVATIONAL COVERING

In this section, we supply details to the pruning strategy *observational covering* used to synthesize the lifting function f. To start, let us consider a trivial enumerator $\mathcal{E}$, which synthesizes by enumerating all functions in G_f from small to large. We denote the enumeration order of $\mathcal{E}$ as $<_e$, where $f_1 <_e f_2$ represents that $\mathcal{E}$ visits function f_1 before f_2. By definition, $<_e$ is a total order on G_f that preserves the order on size, i.e., $f_1 <_e f_2 \Rightarrow \text{size}(f_1) \leq \text{size}(f_2)$. Enumerator $\mathcal{E}$ is inefficient in synthesizing lifting functions as the size of the valid lifting function can be large in practice.

Improving the order. We improve the enumeration order of $\mathcal{E}$ via domain knowledge that a lifting function is usually a tuple of several simpler functions (i.e., in the form of $f_1 \vartriangle \ldots \vartriangle f_k$). Consider two functions $f = f_1 \vartriangle f_2$ and f' where $\text{size}(f)$ is slightly larger than $\text{size}(f')$ but both $\text{size}(f_1)$ and $\text{size}(f_2)$ are much smaller. Since $\text{size}(f') < \text{size}(f)$, enumerator $\mathcal{E}$ prefers f' to f. However, by the domain knowledge, we know that f is more probable to be a valid lifting function than f.

Motivated by this analysis, we relax the order $<_e$ to allow this preference on those large lifting functions that are made up by simple components. Concretely, we consider lifting functions as ordered lists of used components (we denote this form as *composed programs*) and define a relaxed enumeration order as a partial order $\prec_c$ over composed programs.

Definition A.1. A *composed program* $\overline{f}$ is a list $[f_1, \ldots, f_k]$ satisfying (1) each function f_i is in G_f, and (2) $f_k <_e f_{k-1} <_e \cdots <_e f_1$. This composed program corresponds to $f_1 \vartriangle \ldots \vartriangle f_k$.

We say composed program $\overline{f} = [f_1, \ldots, f_k]$ is simpler than $\overline{f'} = [f'_1, \ldots, f'_{k'}]$, denoted as $\overline{f} \prec_c \overline{f'}$, if the components used in $\overline{f}$ is not only fewer but also simpler, i.e., (1) k is no larger than k', and (2) $[f_1, \ldots, f_k]$ is lexicographically smaller than list $[f'_1, \ldots, f'_{k'}]$ in the sense of $<_e$.

On the one hand, the relaxed order $\prec_c$ still prefers simple lifting functions, because $[f_1, \ldots, f_k] \prec_c [f'_1, \ldots, f'_k]$ implies that $\text{size}(f_1 \vartriangle \ldots \vartriangle f_k) \leq \text{size}(f'_1 \vartriangle \ldots f'_{k'})$. On the other hand, $\prec_c$ allows those large functions that are made up by simple components to be visited earlier and thus matches our motivation. For example, for any three functions $f_1 <_e f_2 <_e f_3$, composed program $[f_1, f_2]$ is always allowed to be visited before $[f_3]$ since $[f_3] \not\prec_c [f_1, f_2]$ by definition.

Algorithm 2 shows the pseudocode of an enumerator following the order of $\prec_c$. It maintains a working list (*workingList*) that queues those constructed but not yet visited composed programs and a list (*visitList*) storing all visited composed programs. In each turn, Algorithm 2 obtains the next composed program $\overline{f}$ from either the basic enumerator $\mathcal{E}$ (Line 3) or the working list (Line 4-5). If $\overline{f}$ satisfies all examples, it will be returned as the result (Line 15). Otherwise, $\overline{f}$ will be used to construct new composed programs (Lines 9-14, 20). Note that function `IsNonOptimal` (Line 14) is an interface for *observational covering* and can be regarded as *true* here.

We would like to highlight three noticeable properties of Algorithm 2.

- Compared to the basic enumerator $\mathcal{E}$, Algorithm 2 uniformly explores composed programs with each number of components in range $[1, n_c]$ and thus those lifting functions that are made up by simple components are visited earlier.
- Algorithm 2 is minimal (Definition 5.4) because a function that uses more units is also more complex in the sense of $\prec_c$, and thus it will be visited later in Algorithm 2.
- Algorithm 2 is an Occam solver (Definition 4.9) because only composed programs with at most n_c components are considered. At this time, the visited program is at most n_c times larger than the last program generated by the basic enumerator $\mathcal{E}$[10].

[10]Note that this constraint does not affect the completeness of Algorithm 4.9, because each function $f \in G_f$ can be regarded as composed program $[f]$, which involves only a single component.

Algorithm 2: An improved enumerator for composed programs.

Input: An enumerator $\mathcal{E}$, a set E of examples, and a parameter n_c.
Output: A lifting function f satisfying all examples in E.

```
1  ∀size ≥ 1, workingList[size] ← [], visitedList[size] ← [];
2  Function NextComposedProgram(size):
3      if size = 1 then return [ℰ.Next()] ;
4      if workingList[size].Empty() then return null;
5      return workingList[size].PopFront();
6  Function InsertNewPrograms(size, prog = [f_1, ..., f_size]):
7      visitedList[size].PushBack(prog);
8      for each [f] ∈ visitedList[1] satisfying f ≺_e f_size do
9          workingList[size + 1].PushBack([f_1, ..., f_size, f]);
10     end
11 for turn ← 1 ... + ∞ do
12     s ← (turn − 1) mod n_c + 1;
13     f̄ = [f_1, ..., f_s] ← NextComposedProgram(s);
14     if f̄ = null ∨ IsNonOptimal(size, f̄, E) then continue;
15     if SatisfiedExamples(f̄, E) = E then return f_1 △ ... △ f_s;
16     InsertNewPrograms(s, f̄);
17 end
```

Pruning by observational covering. We further improve Algorithm 2 via a pruning strategy namely *observational covering*. Since Algorithm 2 returns only the first visited program satisfying all examples, those composed programs that (1) are later in the enumeration order $\prec_c$ and (2) covers a smaller set of examples can be safely ignored. We define these programs as *observationally covered* in Definition A.2 and show they are well-structured in Lemma A.3.

Definition A.2. Compoased program $\overline{f}$ 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 A.3. *Composed program $\overline{f}$ is observationally covered on $E \Rightarrow \forall \overline{f'} \supseteq \overline{f}, \overline{f'}$ is observationally covered on E, where $\overline{f_1} \supseteq \overline{f_2}$ represents that all components in $\overline{f_2}$ are in $\overline{f_1}$ as well.*

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 uncovered on E. □

By Lemma A.3, whenever an observationally covered program $\overline{f}$ is visited, all those composed programs including $\overline{f}$ can be ignored directly as they must also be covered. Therefore, in Algorithm 2, there is no need to extend an observationally covered program via `InsertNewPrograms`.

Algorithm 3: The implementation of function IsNonOptimal in Algorithm 2 (Line 14).

```
1 Function CheckUnCovered(size, $\overline{f}$, E):
2     if $\exists \overline{f'} \subset \overline{f}, \overline{f'} \notin visitedList$ then return true;
3     return $\exists k \in [1, size], \exists \overline{f'} \in visitedList[k], E|_{\overline{f}} \subseteq E|_{\overline{f'}}$;
```

Algorithm 3 plugs this optimization into Algorithm 2 by implementing `IsNonOptimal` to rule out observationally covered programs. In Line 3, it verifies whether $\overline{f}$ is observationally covered via the definition (Definition A.2). Because this verification is time-consuming, Algorithm 2 uses Lemma A.3, a sufficient condition for observationally covered, to first preclude programs (Line 2)[11].

Properties. We denote the combination of Algorithm 2 and 3 as synthesizer $\mathcal{O}_c$. Theorem A.4 demonstrates that $\mathcal{O}_c$ is sound, complete, and its synthesis result is optimal. Theorem A.5 shows that $\mathcal{O}_c$ is a valid solver to make the usage of *decoupling* in *AutoLifter* effective.

Theorem A.4. *Given a subtask $\#\varphi_f$ and a set E of examples, let S be the set of composed programs satisfying all examples. 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. *Terminality*. Define set S' as the set of lifting functions corresponding to composed programs in S, i.e., $\{f_1 \triangleleft \ldots \triangleleft f_k \mid (f_1, \ldots, f_k) \in S\}$. Clearly, lifting functions in S' must be valid on E. Because the basic enumerator $\mathcal{E}$ returns all programs in G_f from small to large, it can find a valid lifting function in finite time and thus $\mathcal{O}_c$ must terminate.

Validity. Because $\mathcal{O}_c$ returns only when $\overline{f}$ is verified to be valid (Line 19), the result must be valid.

Minimality. We prove the minimality via the following claim.

- **Claim**: Each time when $\overline{f}$ 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 *visitedList*.

When this claim holds, suppose $\overline{f^*}$ is covered. There must be 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 *visitedList* or *workingList*$[k]$.

- When $\overline{f}$ is inside *visitedList*, $\overline{f}$ 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 considered by the main loop in order. Therefore, $\overline{f}$ must have been 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 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 uncovered composed program $\overline{g'} = [g'_1, \ldots, g'_{k'}]$ that is simpler than $\overline{g}$ under $\prec_c$. 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()`, 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

[11] After excluding observationally covered programs, the contents of *visitedList* are exactly those visited uncovered programs. Therefore, predicate $\overline{f'} \notin visitedList$ in Algorithm 3 (Line 2) is equivalent to predicate $\overline{f'}$ is observationally covered.

Algorithm 4: The verifier under the black-box setting in *AutoLifter*.

Input: A random generator $\mathcal{G}$ for examples and a parameter n_0 for the initial number of examples.
Output: Function `Verify` for verifying the correctness of the synthesis result.

```
1 n_s ← n_0; t_s ← 1; examples ←= ∅;
2 Function Verify(Φ = ∃p, ∀x̄, φ(p, x̄), p_c):
3     while size(p_c) > t_s do (n_s, t_s) ← (n_s × 2, t_s × 2);
4     examples ← examples ∪ G.Generate(n_s − |examples|);
5     counterExamples ← {e ∈ examples} | ¬φ(p_c, e)};
6     if counterExamples = ∅ then return null;
7     return any e ∈ counterExamples;
```

former case, $\overline{f}$ has just been inserted into *visitedList*. 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*$[k]$.

(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 *workingList*$[k]$ in the same invocation of `InsertNewPrograms`. Because $\mathcal{E}$ enumerates programs according to $\prec_e$, lifting functions in *visitedList*$[1]$ must be in the order of $\prec_e$. Therefore, $\overline{g'}$ will be inserted to *workingList*$[k]$ before $\overline{g}$.

So far, we prove the claim, and thus prove the minimality of solver $\mathcal{O}_c$. □

Theorem A.5. *$\mathcal{O}_c$ is both a minimal solver and an Occam solver for $\#\varphi_f$ tasks with examples.*

Proof. The minimality is direct from the definition of $\prec_c$. 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 $\mathcal{E}$, (2) $\mathcal{E}$ enumerates programs from small to large, we know that $\forall i \in [1, k], \text{size}(f_k) \leq \text{size}(f^*)$. Therefore, we have the following inequality.

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

where c represents the binary bits used to express the operator $\mathbin{\triangle}$. Because n_c is a constant, this inequality implies that $\mathcal{O}_c$ is a 1-Occam solver with constant $2n_c$. □

In our implementation, we set n_c to 4 by default because the tuple of four components is already enough for most known lifting problems.

B APPENDIX: IMPLEMENTATION

B.1 Verifier

As discussed in Section 6, to obtain the guarantee provided by Occam solvers, the only task is to ensure the number of random examples used to verify the candidate program is enough. As shown in Algorithm 4, our verifier achieves it by iteratively enlarging the number of examples (n_s) with a size limit t_s on the synthesis result. Each time when a candidate result p_c is given, Algorithm 4 first enlarges the number of examples until the size of p_c is within the size limit (Line 3). Then, it

verifies p_c on n_s random examples generated by $\mathcal{G}$ (Line 5) and regards p_c as correct when it passes all these examples (Line 6).

Theorem B.1 demonstrates the probabilistic correctness and the completeness of a CEGIS solver that takes Algorithm 4 as the verifier.

Theorem B.1. *Given example-based solver $\mathcal{S}$, random generator $\mathcal{G}$ for examples, and parameter n_0, let $\mathcal{S}_{cegis}[\mathcal{S}, \mathcal{G}, n_0]$ be the CEGIS solver where (1) the example-based solver is $\mathcal{S}$ and (2) the verifier is Algorithm 4 configured by $\mathcal{G}$ and n_0. The following two claims on $\mathcal{S}_{cegis}[\mathcal{S}, \mathcal{G}, n_0]$ hold.*

- *(Correctness) For any synthesis task T, if $\mathcal{S}_{cegis}[\mathcal{S}, \mathcal{G}, n_0]$ terminates on synthesis task T, the synthesis result p must satisfy the following formula.*

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

  where $err_{\mathcal{G}}(p)$ represents the probability for p to violate a random example generated by $\mathcal{G}$.
- *(Completeness) $\mathcal{S}_{cegis}[\mathcal{S}, \mathcal{G}, n_0]$ must terminate on a realizable task if $\mathcal{S}$ 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 algorithm, we have the following inequality.

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

When $\mathcal{E}_j$ happens, there must be 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.

$$\begin{aligned}
\Pr[\mathcal{E}_j] &\leq \Pr_{e_1,\ldots,e_n \sim \mathcal{G}}\left[\exists k \in [1, m], err_{\mathcal{G}}(p_k) \geq \epsilon \wedge p_k \text{ satisfies } e_1, \ldots, e_n\right] \\
&\leq \sum_{k=1}^{m} \Pr_{e_1,\ldots,e_n \sim \mathcal{G}}\left[err_{\mathcal{G}}(p_k) \geq \epsilon \wedge p_k \text{ satisfies } e_1, \ldots, e_n\right] \\
&\leq \sum_{k=1}^{m} (1-\epsilon)^n \leq 2^j \exp(-\epsilon \cdot j \cdot n_0) \leq \exp(\ln 2 - \epsilon n_0)^j
\end{aligned}$$

Therefore, the generation error can be bounded.

$$\Pr[err_{\mathcal{G}}(p) \geq \epsilon] \leq \sum_{j=1}^{\infty} \Pr[\mathcal{E}(j)] \leq \sum_{j=1}^{\infty} \exp(\ln 2 - \epsilon n_0)^j = \frac{2\exp(-\epsilon n_0)}{1 - 2\exp(-\epsilon n_0)} < 4\exp(-\epsilon n_0)$$

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 (α, β)-Occam solver[12] with constant c, where $\alpha \geq 1, 0 \leq \beta < 1$, and the size of the smallest valid program is s. When the thresholds have been enlarged t times, there will be $n_0 2^t$ examples, and no other examples will be considered unless $size(p_c)$ is larger than 2^t. Consider the following derivation.

$$size(p_c) \leq 2^t \impliedby cs^{\alpha}(n_0 2^t)^{\beta} \leq 2^t \iff 2^{t(1-\beta)} \geq cs^{\alpha} n_0^{\beta} \iff t \geq \left\lceil \frac{\ln c + \alpha \ln s + \beta \ln n_0}{\ln 2 \cdot (1-\beta)} \right\rceil$$

[12] An example-based Occam solver is specified by three constants α, β, and c [Ji et al. 2021]. When the smallest valid program is s and n example are given, the solver ensures that the size of the synthesis result is at most $cs^{\alpha} n^{\beta}$.

The above derivation shows that the set of examples will never change after a threshold is reached. Because the number of possible examples is finite, after a finite number of CEGIS iterations, all examples will be directly provided to the Occam solver. At this time, the candidate program must satisfy all examples, and thus the CEGIS solver must terminate when $\mathcal{S}$ is an Occam solver. □

In our implementation, we let generator $\mathcal{G}$ concentrate on short lists and small integers to avoid arithmetic overflow.

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

Guided by Theorem B.1, we set n_0 to 10^4 by default. 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} .

B.2 Grammars

In this subsection, we show the complete grammars used in *AutoLifter*. Figure 2 shows the grammar G_f used in our implementation, which is the grammar used by *DeepCoder* [Balog et al. 2017]. Note that some operators in G_f are partial. For example, *head* is defined only for non-empty lists. We complete these operators with a dummy output $\perp$ in our implementation and define the output of any operator on $\perp$ as $\perp$.

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

Fig. 2. Grammar G_f used for synthesizing the lifting function f.

Figure 3 shows the grammar G_τ used in our implementation. Based on the grammar used by SyGuS-Comp [Alur et al. 2019], we add operators for accessing tuples ($N_{\mathbb{T}}.i$) and constructing tuples ($S \vartriangle S$) into G_τ. Because dummy value $\perp$ is involved to the output of lifting functions, which is a part of the input of operator τ, we also extend the semantics in G_τ to support $\perp$ by setting the output of any operator on $\perp$ to $\perp$.

Start symbol	S	$\rightarrow$	$N_{\mathbb{Z}} \mid S \vartriangle S$
Integer expr	$N_{\mathbb{Z}}$	$\rightarrow$	IntConst $\mid \perp \mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid ite\ N_{\mathbb{B}}\ N_{\mathbb{Z}}\ N_{\mathbb{Z}} \mid N_{\mathbb{T}}.i$
Tuple expr	$N_{\mathbb{T}}$	$\rightarrow$	Inputs $\mid N_{\mathbb{T}}.i$
Bool expr	$N_{\mathbb{B}}$	$\rightarrow$	$\neg N_{\mathbb{B}} \mid N_{\mathbb{B}} \wedge N_{\mathbb{B}} \mid N_{\mathbb{B}} \vee N_{\mathbb{B}} \mid N_{\mathbb{Z}} \leq N_{\mathbb{Z}} \mid N_{\mathbb{Z}} = N_{\mathbb{Z}}$
Binary Operator	$\oplus$	$\rightarrow$	$+ \mid - \mid \times \mid div$

Fig. 3. Grammar G_τ used for synthesizing the operator τ.

C APPENDIX: PROOFS

C.1 Proofs for Section 4

Lemma C.1 (Lemma 4.3). *The following formula is a sufficient condition of Formula 7.*

$$\forall p^* \in \mathbb{P}_{\leq lim_s},\ \Pr_{\varphi\sim\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \mid \varphi(p^*)] \leq \epsilon$$

Proof. The conditional probability in Formula 7 can be simplified in the following way.

$$\begin{aligned}
&\Pr_{\varphi\sim\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)] \\
&= \Pr_{\varphi\sim\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \wedge \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}\left[\exists p^* \in \mathbb{P}_{\leq lim_s, \varphi(p^*)}\right] \\
&= \sum_{P\in\mathbb{P}_{\leq lim_s}} [P \neq \emptyset] \Pr_{\varphi\in\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \wedge S(\varphi) = P] \Big/ \sum_{P\in\mathbb{P}_{\leq lim_s}} [P \neq \emptyset] \Pr[S(\varphi) = P] \qquad (20) \\
&= \frac{\sum_{P\in\mathbb{P}_{\leq lim_s}} [P \neq \emptyset] \left(\Pr_{\varphi\in\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \wedge S(\varphi) = P] + (|P|-1)\Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P]\right)}{\sum_{P\in\mathbb{P}_{\leq lim_s}} |P| \Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P]} \qquad (21)
\end{aligned}$$

where $[\cdot] : \texttt{Bool} \to \texttt{Int}$ maps *true* and *false* to 1 and 0 respectively, $S(\varphi)$ represents the set of valid programs within the scalability threshold for task φ, i.e., $S(\varphi) := \{p \in \mathbb{P}_{\leq lim_s} \mid \varphi(p)\}$. In Formula 20, we unfold the two probability according to the value of $S(\varphi)$ (denoted as P). In Formula 21, we use inequality $(a+c)/(b+c) > a/b$ when $a, b, c > 0, a < b$.

When $P \neq \emptyset$, let us consider the following claim.

$$\Pr_{\varphi\in\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \wedge S(\varphi) = P] + (|P|-1)\Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P] \leq \sum_{p^*\in P} \Pr_{\varphi\in\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \wedge S(\varphi) = P] \qquad (22)$$

To prove Formula 22, let φ be any task satisfying $S(\varphi) = P$. There are three cases on $\mathcal{S}(\#\varphi)$.

- When $\neg\varphi(\mathcal{S}(\#\varphi))$, $\tilde{\Phi}(\varphi)$ contributes to both sides for $|P|$ times.
- When, $\mathcal{S}(\#\varphi) \in S(\varphi)$, $\tilde{\Phi}(\varphi)$ contributes to both sides for $|P|-1$ times.
- Otherwise, $\tilde{\Phi}(\varphi)$ contributes to the left and right sides for $|P|-1$ and $|P|$ times respectively.

Therefore, Formula 22 holds. By applying Formula 22 to Formula 21, we get the derivation below.

$$\begin{aligned}
&\frac{\sum_{P\in\mathbb{P}_{\leq lim_s}} [P \neq \emptyset] \left(\Pr_{\varphi\in\tilde{\Phi}}[\neg\varphi(\mathcal{S}(\#\varphi)) \wedge S(\varphi) = P] + (|P|-1)\Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P]\right)}{\sum_{P\in\mathbb{P}_{\leq lim_s}} |P| \Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P]} \\
&\leq \sum_{P\in\mathbb{P}_{\leq lim_s}} \sum_{p^*\in P} \Pr_{\varphi\in\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \wedge S(\varphi) = P] \Big/ \sum_{P\in\mathbb{P}_{\leq lim_s}} \sum_{p^*\in P} \Pr_{\varphi\sim\tilde{\Phi}}[S(\varphi) = P] \\
&= \sum_{p^*\in\mathbb{P}_{\leq lim_s}} \Pr_{\varphi\sim\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \wedge \varphi(p^*)] \Big/ \sum_{p^*\in\mathbb{P}_{\leq lim_s}} \Pr_{\varphi\sim\tilde{\Phi}}[\varphi(p^*)] \\
&\leq \max_{p^*\in\mathbb{P}_{\leq lim_s}} \left(\Pr_{\varphi\sim\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \wedge \varphi(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}[\varphi(p^*)]\right) \qquad (23) \\
&= \max_{p^*\in\mathbb{P}_{\leq lim_s}} \Pr_{\varphi\in\tilde{\Phi}}[\mathcal{S}(\#\varphi) \neq p^* \mid \varphi(p^*)] \qquad (24)
\end{aligned}$$

Formula 23 uses the following inequality when $a_i, b_i > 0$ and $0/0$ is defined as 0.

$$\sum_{i=1}^{n} a_i \Big/ \sum_{i=1}^{n} b_i \leq \max_{i=1}^{n}(a_i/b_i)$$

To prove this inequality, let k be the value on the right-hand side, then $a_i \le kb_i$ for any index i. At this time, $\sum a_i \le \sum kb_i = k\sum b_i$, and thus the left-hand side is no larger than k.

The target lemma is direct from Formula 24 and the form of Formula 7. □

Lemma C.2 (Lemma 4.6). *For any constants α, β, Formula 7 is satisfied for constant $\epsilon \ge \alpha\beta$ if the following conditions are satisfied.*

- *For any program $p^* \in \mathbb{P}_{\le lim_s}$, the size of synthesis domain $\mathcal{S}^D(p^*)$ is no larger than α.*
- *For any program $p^* \in \mathbb{P}_{\le lim_s}$ and any other program p in the synthesis domain, i.e., $p \in \mathcal{S}^D(p^*)/\{p^*\}$, the following formula holds.*

$$\Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \mid \varphi(p^*)] \le \beta \tag{25}$$

Proof. By Lemma 4.3, we need only to prove that for any $p^* \in \mathbb{P}_{\le lim_s}$, conditional probability $\Pr_{\varphi\in\tilde{\Phi}}[\mathcal{S}(\#\varphi) \ne p^* \mid \varphi(p^*)]$ is always no larger than $\alpha\beta$. By the definition of the synthesis domain, after $\varphi(p^*)$ (i.e., $\#\varphi(p^*)$) is given, a necessary condition for $\mathcal{S}(\#\varphi) = p$ is p is a valid program for $\#\varphi$ and p is inside synthesis domain $\mathcal{S}^D(p^*)$. Therefore, we have the following derivation.

$$\begin{aligned}\Pr_{\varphi\in\tilde{\Phi}}[\mathcal{S}(\#\varphi) \ne p^* \mid \varphi(p^*)] &\le \Pr_{\varphi\in\tilde{\Phi}}\left[\exists p \in \mathcal{S}^D(p^*)/\{p^*\}, \#\varphi(p) \mid \varphi(p^*)\right]\\ &\le \sum_{p\in\mathcal{S}^D(p^*)/\{p^*\}} \Pr_{\varphi\in\tilde{\Phi}}[\#\varphi(p) \mid \varphi(p^*)]\\ &\le \left|\mathcal{S}^D(p^*)\right| \max_{p\in\mathcal{S}^D(p^*)/\{p^*\}} \Pr_{\varphi\in\tilde{\Phi}}[\#\varphi(p) \mid \varphi(p^*)]\end{aligned} \tag{26}$$

The target lemma is direct from Formula 26. □

Lemma C.3 (Lemma 4.8). *When $\#(\cdot)$ is an independent weaker approximation, the following formula is a sufficient condition of Formula 9.*

$$\Pr_{\varphi'\sim\#\tilde{\Phi}}[\varphi'(p) \mid \varphi'(p^*)] \le \beta \tag{27}$$

Proof. Let us consider the claim that for any two sets $P_\#, P_c \subset \mathbb{P}$, event "programs in $P_\#$ are valid for $\#\varphi$" is independent of event "programs in P_c are valid for φ^c", i.e., the following formula.

$$\Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_\#, \#\varphi(p) \wedge \forall p \in P_c, \varphi^c(p)] = \Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_\#, \#\varphi(p)] \Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_c, \varphi^c(p)]$$

To prove this claim, we enumerate the solution sets of $\#\varphi$ and φ^c (denoted as A and B respectively) and perform the following derivation.

$$\begin{aligned}\Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_\#, \#\varphi(p) \wedge \forall p \in P_c, \varphi^c(p)] &= \sum_{P_\#\subseteq A\subseteq\mathbb{P}} \sum_{P_c\subseteq B\subseteq\mathbb{P}} \Pr_{\varphi\sim\tilde{\Phi}}[\mathbb{P}_{\#\varphi} = A \wedge \mathbb{P}_{\varphi^c} = B]\\ &= \sum_{P_\#\subseteq A\subseteq\mathbb{P}} \sum_{P_c\subseteq B\subseteq\mathbb{P}} \Pr_{\varphi\sim\tilde{\Phi}}[\mathbb{P}_{\#\varphi} = A] \Pr_{\varphi\sim\tilde{\Phi}}[\mathbb{P}_{\varphi^c} = B]\\ &= \left(\sum_{P_\#\subseteq A\subseteq\mathbb{P}} \Pr_{\varphi\sim\tilde{\Phi}}[\mathbb{P}_{\#\varphi} = A]\right)\left(\sum_{P_c\subseteq B\subseteq\mathbb{P}} \Pr_{\varphi\sim\tilde{\Phi}}[\mathbb{P}_{\varphi^c} = B]\right)\\ &= \Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_\#, \#\varphi(p)] \Pr_{\varphi\sim\tilde{\Phi}}[\forall p \in P_c, \varphi^c(p)]\end{aligned} \tag{28}$$

where Formula 28 uses the fact that $\#\varphi$ is an independently weaker approximation.

Using this claim, we transform the conditional probability in Formula 9 as the following.

$$\begin{aligned}\Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \mid \varphi(p^*)] &= \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \wedge \varphi(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}[\varphi(p^*)] \\ &= \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \wedge \#\varphi(p^*) \wedge \varphi^c(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p^*) \wedge \varphi^c(p^*)] \\ &= \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \wedge \#\varphi(p^*)] \Pr_{\varphi\sim\tilde{\Phi}}[\varphi^c(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p^*)] \Pr_{\varphi\sim\tilde{\Phi}}[\varphi^c(p^*)] \\ &= \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \wedge \#\varphi(p^*)] \Big/ \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p^*)] = \Pr_{\varphi\sim\tilde{\Phi}}[\#\varphi(p) \mid \#\varphi(p^*)]\end{aligned}$$

The target lemma is direct from the last formula. □

Lemma C.4 (Lemma 4.10). *For any α-Occam solver $\mathcal{S}$ with constant c and any program $p^* \in \mathbb{P}$, the following inequalities always hold.*

$$\mathcal{S}^D(p^*) \subseteq \mathbb{P}_{\leq csize(p^*)^\alpha} \qquad \left|\mathcal{S}^D(p^*)\right| \leq |\mathbb{P}_{\leq csize(p^*)^\alpha}| \leq 2^{csize(p^*)^\alpha}$$

Proof. This lemma is direct from the definition of Occam solvers. □

Theorem C.5 (Theorem 4.11). *The error rate of an α-Occam solver $\mathcal{S}$ with constant c under an independently weaker approximation $\#(\cdot)$ is bounded by $2^{c lim_s^\alpha}\beta$ for constant β, if the following condition holds.*

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \forall p \in \mathcal{S}^D(p^*)/\{p^*\}, \Pr_{\varphi'\sim\#\tilde{\Phi}}[\varphi'(p) \mid \varphi'(p^*)] \leq \beta \tag{29}$$

Proof. This theorem is direct from Lemma 4.6, 4.8 and 4.10. □

Theorem C.6 (Theorem 4.13). *Let $\mathcal{S}$ be a probabilistic α-Occam solver $\mathcal{S}$ with constant (c, γ) for $\#\tilde{\Phi}_{lim}$, the induced distribution of concerned tasks, i.e.,*

$$\#\tilde{\Phi}_{lim}(\varphi') = \Pr_{\phi\sim\tilde{\Phi}}[\varphi' = \#\varphi \mid \exists p \in \mathbb{P}_{\leq lim_s}, \varphi(p)]$$

The error rate of $\mathcal{S}$ under an independently weaker approximation $\#(\cdot)$ is bounded by $2^{c lim_s^\alpha}\beta + \gamma$ with constant β, if the following condition holds.

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \forall p \in \left(\mathcal{S}^D(p^*) \cup \mathbb{P}_{\leq size(p^*)}\right)/\{p^*\}, \Pr_{\varphi'\sim\#\tilde{\Phi}}[\varphi'(p) \mid \varphi'(p^*)] \leq \beta \tag{30}$$

Proof. Define solver $\mathcal{S}_o$ as the following, where p_{min} is the smallest valid program for task $\#\varphi$. Specially, when $\#\varphi$ is unrealizable, p_{min} is defined as $\perp$.

$$\mathcal{S}_o(\#\varphi) = \begin{cases} \mathcal{S}(\#\varphi) & \mathcal{S}(\#\varphi) \neq \perp \wedge size(\mathcal{S}(\#\varphi)) \leq csize(p^*)^\alpha \\ p_{min} & \text{Otherwise} \end{cases}$$

Clearly, $\mathcal{S}_o$ is an α-Occam solver with constant c. Consider a program p in synthesis domain $\mathcal{S}_o^D(p^*)$, there are two possible cases.

- When p is the synthesis result given by $\mathcal{S}$, p must be in synthesis domain $\mathcal{S}^D(p^*)$.
- When p is the smallest valid solution p_{min}, p must be no larger than p^*, i.e., in set $\mathbb{P}_{\leq size(p^*)}$.

Therefore, $\mathcal{S}_o^D(p^*) \subseteq \mathcal{S}^D(p^*) \cup \mathbb{P}_{\leq size(p^*)}$ holds. By Formula 30, we have the following formula.

$$\forall p^* \in \mathbb{P}_{\leq lim_s}, \forall p \in \mathcal{S}_o^D(p^*)/\{p^*\}, \Pr_{\varphi'\sim\#\tilde{\Phi}}[\varphi'(p) \mid \varphi'(p^*)] \leq \beta$$

By Theorem 4.11, the error rate of $\mathcal{S}_o$ is bounded, as shown below.

$$\Pr_{\varphi\sim\tilde{\Phi}}[\neg\varphi(\mathcal{S}_o(\#\varphi)) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)] \leq 2^{c lim_s^\alpha}\beta$$

At last, we prove the target theorem by the following derivation.

$$\begin{aligned}
&\Pr_{\varphi\sim\tilde{\Phi}}\left[\neg\varphi(\mathcal{S}(\#\varphi)) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)\right] \\
\leq\ &\Pr_{\varphi\sim\tilde{\Phi}}\left[\mathcal{S}(\#\varphi) \neq \mathcal{S}_o(\#\varphi) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)\right] + \Pr_{\varphi\sim\tilde{\Phi}}\left[\neg\varphi(\mathcal{S}_o(\#\varphi)) \mid \exists p^* \in \mathbb{P}_{\leq lim_s}, \varphi(p^*)\right] \\
\leq\ &\gamma + 2^{clim_s^{\alpha}}\beta
\end{aligned}$$

□

C.2 Proofs for Section 5

Lemma C.7 (Lemma 5.1). *For any generalized lifting problem φ_g that is realizable, decomposition guarantees that (soundness) the constructed program must be valid for φ_g, and (completeness) the first subtask and all possible second subtasks must be realizable.*

Proof. For soundness, let (f_1, τ_1) and (f_2, τ_2) be any valid program for $\varphi_{g,1}$ and $\varphi_{g,2}[f_1, \tau_1]$ respectively. We verify the validity of $(f_1 \vartriangle f_2, (\tau_1 \circ \sigma_1) \vartriangle (\tau_2 \circ \sigma_2))$ for task $\varphi_{g,1}$ as the following.

$$\begin{aligned}
&((\tau_1 \circ \sigma_1) \vartriangle (\tau_2 \circ \sigma_2))\ \left(x, (h \vartriangle (f_1 \vartriangle f_2))^n\ \overline{a}\right) \\
=\ &\left((\tau_1 \circ \sigma_1)\ \left(x, (h \vartriangle (f_1 \vartriangle f_2))^n\ \overline{a}\right), (\tau_2 \circ \sigma_2)\ \left(x, (h \vartriangle (f_1 \vartriangle f_2))^n\ \overline{a}\right)\right) \\
=\ &\left(\tau_1\ \left(x, (h \vartriangle f_1)^n\ \overline{a}\right), \tau_2\ \left(x, ((h \vartriangle f_1) \vartriangle f_2)^n\ \overline{a}\right)\right) \\
=\ &\left(p\ (\mu\ (x, \overline{a})), f\ (\mu\ (x, \overline{a}))\right) \\
=\ &(p \vartriangle f)\ (\mu\ (x, \overline{a}))
\end{aligned}$$

For the completeness, let $\varphi_{g,1}$ be a realizable task with valid program (f, τ). Because the output of τ is a tuple, τ must be in the form of $\tau_1 \triangle \tau_2$ such that the following formulas are satisfied.

$$\begin{aligned}
&p\ (\mu\ (x, \overline{a})) = \tau_1\ \left(x, (h \vartriangle f)^n\ \overline{a}\right) \\
&f\ (\mu\ (x, \overline{a})) = \tau_2\ \left(x, (h \vartriangle f)^n\ \overline{a}\right) \\
\Longrightarrow\ &(f \vartriangle null)\ (\mu\ (x, \overline{a})) = ((\tau_2 \circ \sigma) \vartriangle null)\ \left(x, ((h \vartriangle f) \vartriangle null)^n\ \overline{a}\right)
\end{aligned}$$

where σ is a function reorganizing the input for τ_2. Therefore, (f, τ_1) is a valid solution for $\varphi_{g,1}$, i.e., $\varphi_{g,1}$ is realizable. The realizability of the second subtask $\varphi_{g,2}[f_1, \tau_1]$ has been encoded in the specification of the first subtasks $\varphi_{g,1}$. □

Theorem C.8 (Theorem 5.2). *For any scalability threshold lim_s, any limits N and L over the size of integers and lists, and any α-Occam solver $\mathcal{S}$ with constant c, the error rate of $\#\varphi_{g,1}$ and $\mathcal{S}$ under model $\tilde{\Phi}_{g,1}$ is bounded by the following formula.*

$$2^{clim_s^{\alpha}}\left(N^{-N+1} + N\exp\left(-N^{L-2}\right)\right)$$

Proof. By Theorem 4.11, we only need to prove the following claim, where the size of f_1^*, τ_1^* is no more than lim_s, and (f_1, τ_1) is in the corresponding synthesis domain of $\mathcal{S}$.

$$\Pr_{\varphi_{g,1}\sim\tilde{\Phi}_{g,1}}\left[\#\varphi_{g,1}(f_1, \tau_1) \mid \#\varphi_{g,1}(f_1^*, \tau_1^*)\right] \leq N^{-N+1} + N\exp\left(-N^{L-1}\right) \tag{31}$$

We first simplify the conditional probability in Formula 31 as the following. For simplicity, we omit the signature of the randomness (i.e., $\varphi_{g,1} \sim \tilde{\Phi}_{g,1}$) and the quantifiers corresponding to $x, \overline{a}$ in

the definition of $\#\varphi_{g,1}$ (i.e., $\forall x \in C, \forall \overline{a} \in A^n$).

$$\begin{aligned}
&\Pr\left[\#\varphi_{g,1}(f_1,\tau_1) \mid \#\varphi_{g,1}(f_1^*,\tau_1^*)\right] \\
&= \Pr\left[p\ (\mu\ (x,\overline{a})) = \tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) \,\middle|\, p\ (\mu\ (x,\overline{a})) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right] \\
&= \frac{\Pr\left[p\ (\mu\ (x,\overline{a})) = \tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right]}{\Pr\left[p\ (\mu\ (x,\overline{a})) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right]} \\
&= \frac{\Pr\left[p\ (\mu\ (x,\overline{a})) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right]\Pr\left[\tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right]}{\Pr\left[p\ (\mu\ (x,\overline{a})) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right]} \qquad (32) \\
&= \Pr\left[\tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_1^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right] \qquad (33)
\end{aligned}$$

where Formula 32 uses (1) the independence between $(p \circ \mu)$ and $(\tau_1, \tau_1^*, f_1, f_1^*, h)$, and (2) the complete randomness of $p \circ \mu$.

To analyze the probability involved in Formula 33, there are two possible cases.

Case 1: $\tau_1 \neq \tau_1^*$. Let $\tau_{1,i}$ and $\tau_{1,i}^*$ be the ith unit used in τ and τ_1^* respectively. In this case, there must be some index k such that $\tau_{1,k} \neq \tau_{1,k}^*$. Because $\tau_1\ w = \tau_1^* w \Rightarrow \tau_{1,k}\ w = \tau_{1,k}^*\ w$, the following probability is no smaller than the probability involved in Formula 33.

$$\Pr\left[\tau_{1,k}\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_{1,k}^*\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right)\right] \qquad (34)$$

Let I_τ be the set of involved inputs of $\tau_{1,k}$, i.e., $\{(x,(h \vartriangle f_1)^n\ \overline{a})\}$. Because for each input $\bar{i}$ in I_τ, the probability in Formula 34 requires $\tau_{1,k}\ \bar{i}$ to be some output generated by $\tau_{1,k}^*$, a function independent of $\tau_{1,k}$. Therefore, this probability is bounded by $\Pr[|I_\tau| < N] + N^{-N}$.

Let O_h be the range of function h, i.e., $\{h\ a \mid a \in A\}$. Because the output of h is a component in the input of $\tau_{1,k}$, event $|I_\tau| < N$ implies event $|O_h| < N$. Using this relation, we bound the probability of $|I_\tau| < N$ as the following.

$$\begin{aligned}
\Pr[|I_\tau| < N] &\leq \Pr[|O_h| < N] \\
&= \Pr[\exists i \in [1,N], \forall a \in A, h\ a \neq i] \\
&\leq \sum_{i=1}^{N} \Pr[\forall a \in A, h\ a \neq i] \\
&\leq N(1-1/N)^{|A|} \leq N \exp\left(-N^{L-1}\right)
\end{aligned}$$

Therefore, in the first case, the probability in Formula 33 is no more than $N^{-N} + N\exp(-N^{L-1})$.

Case 2: $\tau_1 = \tau_1^* \wedge f_1 \neq f_1^*$. Let $f_{1,i}$ and $f_{1,i}^*$ be the ith unit used in f_1 and f_1^* respectively. In this case, there must be some index k such that $f_{1,k} \neq f_{1,k}^*$. Suppose the choices of $f_{1,k}$ and $f_{1,k}^*$ are determined before all the other black-box functions, and define event $\mathcal{E}_f$ as "for each $i \in [1, N-1]$, there is an $a_i \in A$ such that $f_{1,k}\ a_i = i$ and $f_{1,k}\ a^i = i+1$". As the first step, We prove the concerned probability in Formula 33 is bounded when $\mathcal{E}_f$ is satisfied, i.e., the probability below is bounded.

$$\Pr\left[\tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_1\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right) \,\middle|\, \mathcal{E}_f\right]$$

Because $\mathcal{E}_f$ relies only on $f_{1,k}$ and $f_{1,k}^*$, we need only to prove that for any choices of $f_{1,k}$ and $f_{1,k}^*$ (denoted as g and g^* respectively) satisfying $\mathcal{E}_f$, the following probability is always small.

$$\Pr\left[\tau_1\ \left(x,(h \vartriangle f_1)^n\ \overline{a}\right) = \tau_1\ \left(x,(h \vartriangle f_1^*)^n\ \overline{a}\right) \,\middle|\, f_{1,k} = g \wedge f_{1,k}^* = g^*\right] \qquad (35)$$

Let $\overline{a_i}$ be any value in A^n such that the first component is equal to a^i, and let x_0 be any value in C. Consider the following claim on several inputs of τ_1.

$$\left(x_0, (h \vartriangleleft f_1^*)^n\ \overline{a_i}\right) \notin \left\{\left(x, (h \vartriangleleft f_1)^n\ \overline{a_j}\right) \,\middle|\, \forall j \in [1,i]\right\} \cup \left\{\left(x, (h \vartriangleleft f_1^*)^n\ a_j\right) \,\middle|\, \forall j \in [1,i-1]\right\} \tag{36}$$

For each value in Formula 35, consider the position corresponding to the first component in $\overline{a_i}$ (or $\overline{a_j}$) and $f_{1,k}$ (or $f_{1,k}^*$). By the construction of $\overline{a_i}$, on the left-hand side, the number on that position is $i+1$. In contrast, on the right-hand side, the numbers on that position is at most i. Therefore, Formula 36 must hold.

By the above claim, we bound the probability in Formula 35 as the following, where event $\mathcal{E}_{f,i}$ is defined as $(x_0, (h \vartriangleleft f_1)^n\ \overline{a_i}) = \tau_1\ (x_0, (h \vartriangleleft f_1^*)^n\ \overline{a_i})$.

$$\begin{aligned}
&\Pr\left[\tau_1\left(x, (h \vartriangleleft f_1)^n\ \overline{a}\right) = \tau_1\left(x, (h \vartriangleleft f_1^*)^n\ \overline{a}\right) \,\middle|\, f_{1,k} = g \wedge f_{1,k}^* = g^*\right] \\
\le\ &\Pr\left[\bigwedge_{i=1}^{N-1} \mathcal{E}_{f,i} \,\middle|\, f_{1,k} = g \wedge f_{1,k}^* = g^*\right] \\
\le\ &\prod_{i=1}^{N-1} \Pr\left[\mathcal{E}_{f,i} \,\middle|\, f_{1,k} = g \wedge f_{1,k}^* = g^* \wedge \bigwedge_{j=1}^{i-1} \mathcal{E}_{f,j}\right]
\end{aligned} \tag{37}$$

By Formula 36, $\mathcal{E}_{f,1}, \ldots, \mathcal{E}_{f,i-1}$ never refers to the output of τ_1 on $(x_0, (h \vartriangleleft f_1^*)^n\ \overline{a_i})$. At this time, each term in Formula 37 asks for the probability for $\tau_1\ (x_0, (h \vartriangleleft f_1^*)^n\ \overline{a_i})$, an integer uniformly drawn from $[1,N]$, to be equal to $\tau_1\ (x_0, (h \vartriangleleft f_1)^n\ \overline{a_i})$, an independent integer drawn from some other distribution. Therefore, each of them is equal to N^{-1} and Formula 37 is equal to N^{-N+1}.

Now we term to our target, the probability specified in Formula 33. By the above discussion on event $\mathcal{E}_f$, we bound the concerned probability as the following.

$$\begin{aligned}
&\Pr\left[\tau_1\left(x, (h \vartriangleleft f_1)^n\ \overline{a}\right) = \tau_1^*\left(x, (h \vartriangleleft f_1^*)^n\ \overline{a}\right)\right] \\
\le\ &\Pr[\neg\mathcal{E}_f] + \Pr\left[\tau_1\left(x, (h \vartriangleleft f_1)^n\ \overline{a}\right) = \tau_1^*\left(x, (h \vartriangleleft f_1^*)^n\ \overline{a}\right) \,\middle|\, \mathcal{E}_f\right] \\
\le\ &\Pr\left[\exists i \in [1,N-1], \forall a \in A, \left(f_{1,k}\ a \neq i \vee f_{1,k}^*\ a \neq i+1\right)\right] + N^{-N+1} \\
\le\ &\sum_{i=1}^{N-1} \Pr\left[\forall a \in A, \left(f_{1,k}\ a \neq i \vee f_{1,k}^*\ a \neq i+1\right)\right] + N^{-N+1} \\
=\ &(N-1)(1-1/N^2)^{|A|} + N^{-N+1} \le N^{-N+1} + N\exp(-N^{L-2})
\end{aligned}$$

The target theorem is directly by summing the results of the two cases up. □

Lemma C.9 (Lemma 5.3). *For any synthesis task in the form of $\#\varphi_{g,1}$ that is realizable, decoupling guarantees that (soundness) the constructed program must be valid for $\#\varphi_{g,1}$, and (completeness) the first subtask and all possible second subtask must be realizable.*

Proof. This lemma is direct from the definition of subtasks. □

Theorem C.10 (Theorem 5.5). *For any scalability threshold lim_s, any limits N, L, any α-Occam solver $\mathcal{S}$ with constant c that is also minimal, the error rate of $\#\varphi_f$ and $\mathcal{S}$ under model $\tilde{\Phi}_f$ is bounded by the following formula.*

$$2^{c'\,lim_s^\alpha} \times 2\left(N^{-N} + N^s \exp\left(-N^{L-s}\right)\right), \text{ where } s = 1 + lim_s + clim_s^\alpha$$

Proof. As discussed in the main text, we use $U(f)$ to denote the set of units used in lifting function f and regards two lifting functions formed by the same set of units as the same. By Theorem 4.11, we only need to bound the following probability, where the size of f_1^* is no more

than lim_s, and f_1 is in the corresponding synthesis domain of $\mathcal{S}$. For simplicity, we omit the signature of the randomness (i.e., $\varphi_f \sim \tilde{\Phi}_f$) and the quantifiers corresponding to $x, \overline{a}, \overline{a'}$ in the definition of $\#\varphi_f$ (i.e., $\forall x \in C, \forall \overline{a}, \overline{a'} \in A^n$).

$$\begin{aligned}\Pr[&(h \vartriangleleft f_1)^n\ \overline{a} = (h \vartriangleleft f_1)^n\ \overline{a'} \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\\ &\mid (h \vartriangleleft f_1^*)^n\ \overline{a} = (h \vartriangleleft f_1^*)^n\ \overline{a'} \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))]\end{aligned}$$

By the definition of minimal solvers, $U(f_1)$ must not be a super set of $U(f_1^*)^*$. Let f, g, g^* be the lifting functions formed by units in $U(f_1) \cup U(f_1^*), U(f_1)/U(f_1^*), U(f_1^*)/U(f_1)$, respectively. Then (1) g^* must not be a constant function, (2) f_1 and f_1^* can be regarded as $f \vartriangleleft g$ and $f \vartriangleleft g^*$ respectively, and (3) f, g, g^* are completely independent.

We transform the target conditional probability as the following.

$$\begin{aligned}&\Pr[(h \vartriangleleft f_1)^n\ \overline{a} = (h \vartriangleleft f_1)^n\ \overline{a'} \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\\ &\qquad \mid (h \vartriangleleft f_1^*)^n\ \overline{a} = (h \vartriangleleft f_1^*)^n\ \overline{a'} \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))]\\ =\ &\Pr\Big[\Big((h \vartriangleleft f)^n\ \overline{a} = (h \vartriangleleft f)^n\ \overline{a'} \wedge g^n\ \overline{a} = g^n\ \overline{a'}\Big) \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\\ &\qquad \Big|\ \Big((h \vartriangleleft f)^n\ \overline{a} = (h \vartriangleleft f)^n\ \overline{a'} \wedge g^{*n}\ \overline{a} = g^{*n}\ \overline{a'}\Big) \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\Big]\\ =\ &\Pr\Big[\Big((h \vartriangleleft f)^n\ \overline{a} = (h \vartriangleleft f)^n\ \overline{a'} \wedge \Big(g^n\ \overline{a} = g^n\ \overline{a'} \vee g^{*n}\ \overline{a} = g^{*n}\ \overline{a'}\Big)\Big) \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\Big]\\ &\qquad \Big/ \Pr\Big[(h \vartriangleleft f \vartriangleleft g^*)^n\ \overline{a} = (h \vartriangleleft f \vartriangleleft g^*)^n\ \overline{a'} \to p\ (\mu\ (x, \overline{a})) = p\ (\mu\ (x, \overline{a'}))\Big] \qquad (38)\end{aligned}$$

Let A and B be the events in the numerator and the denominator of Formula 38 respectively. To bound the value of Formula 38, we need to find an upper-bound for $\Pr[A]$ and a lower-bound for $\Pr[B]$. We start with a claim on surjective functions.

Claim 1. Let f be a random function from lists to n_f integers. The probability for f to be a surjective function is at least $1 - N^{n_f} \exp(-N^{L-n_f})$.

Proof of Claim 1. Let $\mathcal{E}_{sur}$ be the event that f is surjective. We bound $\Pr[\neg\mathcal{E}_{sur}]$ as the following.

$$\Pr[\neg\mathcal{E}_{sur}] = \Pr[\exists o, \forall i, f\ i \neq o] \leq N^{n_f}(1 - N^{-n_f})^{N^L} \leq N^{n_f} \exp(-N^{L-n_f})$$

Therefore, the probability for f to be surjective is at least $1 - N^{n_f} \exp(-N^{L-n_f})$.

Lower-bound for $\Pr[B]$. We define a context (written as $\mathcal{C}$) as an assignment to black-box functions except $p \circ m$. For each context $\mathcal{C}$, we construct a graph $G^B_{full}[\mathcal{C}]$ as the following.

- For each $x \in C, \overline{a} \in A^n$, add a vertex $(x, \overline{a})$ to $G^B_{full}[\mathcal{C}]$.
- For any two vertices $(x, \overline{a})$ and $(x, \overline{a'})$, add an undirected edge between them to $G^B_{full}[\mathcal{C}]$ if the premise of event B is satisfied by context $\mathcal{C}$, i.e., $(h \vartriangleleft f \vartriangleleft g^*)^n\ \overline{a} = (h \vartriangleleft f \vartriangleleft g^*)^n\ \overline{a'}$.

By the definition of event B, it happens under context $\mathcal{C}$ if and only if for any connected component in $G^B_{full}[\mathcal{C}]$, function $(p \circ \mu)$ outputs the same on all inputs corresponding to the vertices in the component. Therefore, the following equality holds, where $\#cc(G)$ represents the number of connected components in graph G.

$$\Pr[B \mid \mathcal{C}] = N^{-|C||A|^n} N^{\#cc(G^B_{full}[\mathcal{C}])}$$

To deal with $\#cc(G^B_{full}[\mathcal{C}])$, we construct another graph $G_B[\mathcal{C}]$, which consider only a single list.

- For each $a \in A$, add a vertex a to $G_B[\mathcal{C}]$.
- For any two vertices a and a', add an undirected edge between them to $G_B[\mathcal{C}]$ if $(f \vartriangleleft f \vartriangleleft g^*)$ output the same on them, i.e., $(h \vartriangleleft f \vartriangleleft g^*)\ a = (h \vartriangleleft f \vartriangleleft g^*)\ a'$.

We make the following claim on the numbers of connected components of $G^B_{full}[\mathcal{C}]$ and $G_B[\mathcal{C}]$.

Claim 2. For any context $\mathcal{C}$, $\#cc(G^B_{full}[\mathcal{C}]) = |C|\#cc(G_B[\mathcal{C}])^n$.

Proof of Claim 2. By the definition of $G^B_{full}[\mathcal{C}]$, two vertices are in the same connected component if and only if they are directly connected by an edge. The same property also holds for $G_B[\mathcal{C}]$.

For any two vertices $(x, \overline{a})$ and $(x', \overline{a'})$ in $G^B_{full}[\mathcal{C}]$, they are directly connected in $G^B_{full}[\mathcal{C}]$ if and only if $x = x'$ and for each $i \in [1, n]$ function $(h \vartriangleleft f \vartriangleleft g^*)$ outputs the same on the ith components in $\overline{a}$ and $\overline{a'}$ (written as $\overline{a}[i]$ and $\overline{a'}[i]$), i.e., $\overline{a}[i]$ and $\overline{a'}[i]$ are directly connected in $G_B[\mathcal{C}]$.

Therefore, there is a bijection between the connected components in $G^B_{full}[\mathcal{C}]$ and the tuple of a value in C and n components in $G_B[\mathcal{C}]$, which directly implies the target equation.

The following formulas are direct from Claim 2.

$$\Pr[B \mid \mathcal{C}] = N^{-|C||A|^n} N^{\#cc(G^B_{full}[\mathcal{C}])} = N^{-|C||A|^n} N^{|C|\#cc(G_B[\mathcal{C}])^n} \tag{39}$$

$$\Pr[B \mid \mathcal{C}] \le N^{-|C||A|^n} N^{|C|N^{mn}} \tag{40}$$

where m represents the number of integers in the output of $(h \vartriangleleft f \vartriangleleft g^*)$.

Let $\mathcal{E}_B$ be the event that $(h \vartriangleleft f \vartriangleleft g^*)$ is surjective. Clearly, when context $\mathcal{C}$ makes $\mathcal{E}_B$ happen, $\#cc(G_B[\mathcal{C}])$ is equal to N^m. Using this property, we derive a lower-bound for $\Pr[B]$ as follows.

$$\begin{aligned}
\Pr[B] &\ge \Pr[B, \mathcal{E}_B] = \Pr[\mathcal{E}_B] \Pr[B \mid \mathcal{E}_B] \\
&\ge \left(1 - N^m \exp\left(-N^{L-m}\right)\right) \Pr[B \mid \mathcal{E}_B] && (41) \\
&= \left(1 - N^m \exp\left(-N^{L-m}\right)\right) N^{-|C||A|^n} N^{|C|N^{mn}} && (42)
\end{aligned}$$

where Formula 41 and 42 are from *Claim 1* and Formula 39 respectively.

Upper-bound for $\Pr[A]$. Similarly, for each context $\mathcal{C}$, we construct a graph $G^A_{full}[\mathcal{C}]$ as follows.

- For each $x \in C, \overline{a} \in A^n$, add a vertex $(x, \overline{a})$ to $G^A_{full}[\mathcal{C}]$.
- For any two vertices $(x, \overline{a})$ and $(x, \overline{a'})$, add an undirected edge between them to $G^A_{full}[\mathcal{C}]$ if the premise of event A is satisfied by context $\mathcal{C}$, i.e.,

$$(h \vartriangleleft f)^n\ \overline{a} = (h \vartriangleleft f)^n\ \overline{a'} \wedge \left(g^n\ \overline{a} = g^n\ \overline{a'} \vee g^{*n}\ \overline{a} = g^{*n}\ \overline{a'}\right)$$

The conditional probability $\Pr[A \mid \mathcal{C}]$ is also related to the number of connected components in G^A_{full}, as shown in the following equality.

$$\Pr[A \mid \mathcal{C}] = N^{-|C||A|^n} N^{\#cc(G^A_{full}[\mathcal{C}])}$$

Compared graph $G^A_{full}[\mathcal{C}]$ to $G^B_{full}[\mathcal{C}]$, we can find that each edge in $G^A_{full}[\mathcal{C}]$ must also be in $G^B_{full}[\mathcal{C}]$. Therefore, $\#cc(G^A_{full}[\mathcal{C}]) \le \#cc(G^B_{full}[\mathcal{C}])$, which implies the following inequality.

$$\Pr[A \mid \mathcal{C}] \le \Pr[B \mid \mathcal{C}] \le N^{-|C||A|^n} N^{|C|N^{mn}} \tag{43}$$

where the last inequality is from Formula 40.

Let $\mathcal{E}_B$ be the event that function $h \vartriangleleft f \vartriangleleft g^* \vartriangleleft g$ is surjective. Consider the following claim.

Claim 3. When context $\mathcal{C}$ makes $\mathcal{E}_A$ happen, $\#cc(G^A_{full}[\mathcal{C}])$ is equal to $|C|N^{nm'}$, where m' represents the number of integers in the output of $h \vartriangleleft f$.

Proof of Claim 3. Because $(h \vartriangleleft f)$ is given to be surjective, vertices in $G^A_{full}[\mathcal{C}]$ can be divided into $|C|N^{nm'}$ non-empty categories according to x and $(h \vartriangleleft f)^n\ \overline{a}$. By the definition of $G^A_{full}[\mathcal{C}]$, all edges in the graph connect two vertices from the same category. Therefore, to get the target claim, we need only to prove that all categories are connected.

Let $(x, \overline{a})$ and $(x, \overline{a'})$ be any vertices in the same category, i.e., $(h \triangleleft f)^n\ \overline{a} = (h \triangleleft f)^n\ \overline{a'}$. Let $\overline{a_m}$ be any other value in A^n such that (1) $(h \triangleleft f)^n\ \overline{a} = (h \triangleleft f)^n\ \overline{a_m}$, (2) $g^n\ \overline{a} = g^n\ \overline{a_m}$, and (3) $g^{*n}\ \overline{a'} = g^{*n}\ \overline{a_m}$. Because $h \triangleleft f \triangleleft g^* \triangleleft g$ is given to be surjective, such a value $\overline{a_m}$ must exist. At this time, there is an edge between $(x, \overline{a})$, $(x, \overline{a_m})$, and there is also an edge between $(x, \overline{a_m})$, $(x, \overline{a'})$. Therefore, $(x, \overline{a})$ and $(x, \overline{a'})$ must be connected, which implies the target claim.

By Claim 3, we derive an upper-bound for $\Pr[A]$ as the following.

$$\begin{aligned}
\Pr[A] &\leq \Pr[A \mid \mathcal{E}_A] + \Pr[\neg\mathcal{E}_A]\Pr[A \mid \neg\mathcal{E}_A] \\
&= N^{-|C||A|^n} N^{|C|N^{m'n}} + \Pr[\neg\mathcal{E}_A]\Pr[A \mid \neg\mathcal{E}_A] && (44)\\
&\leq N^{-|C||A|^n} N^{|C|N^{m'n}} + \Pr[\neg\mathcal{E}_A] N^{-|C||A|^n} N^{|C|N^{mn}} && (45)\\
&\leq N^{-|C||A|^n} N^{|C|N^{m'n}} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right) N^{-|C||A|^n} N^{|C|N^{mn}} && (46)
\end{aligned}$$

where n_g represents the number of integers in the output of g, Formula 44, 45, and 46 are from Claim 3, Formula 43, and Claim 1 respectively.

Bound the target probability. Using the derived upper-bound for $\Pr[A]$ and lower-bound for $\Pr[B]$, we bound the target probability, which is equal to $\Pr[A]/\Pr[B]$, by the following derivation.

$$\begin{aligned}
\frac{\Pr[A]}{\Pr[B]} &\leq \frac{N^{-|C||A|^n} N^{|C|N^{m'n}} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right) N^{-|C||A|^n} N^{|C|N^{mn}}}{\left(1 - N^m \exp\left(-N^{L-m}\right)\right) N^{-|C||A|^n} N^{|C|N^{mn}}} \\
&= \frac{N^{|C|(N^{m'n} - N^{mn})} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right)}{\left(1 - N^m \exp\left(-N^{L-m}\right)\right)} \\
&\leq \frac{N^{-N^{m-m'}} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right)}{\left(1 - N^m \exp\left(-N^{L-m}\right)\right)} && (47)
\end{aligned}$$

The following formulas bounds the constants involved in the above formula, where n_f represents the number of integers returned by function f.

$$m - m' = n_{g^*} \leq 1 \quad m = n_h + (n_f + n_{g^*}) \leq 1 + size(f_1^*) \leq 1 + lim_s$$
$$n_g \leq size(f_1) \leq csize(f_1^*)^\alpha \leq clim_s^\alpha$$

Based on these inequalities, we can further simplify Formula 47 as the following.

$$\frac{\Pr[A]}{\Pr[B]} \leq \frac{N^{-N} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right)}{\left(1 - N^m \exp\left(-N^{L-m}\right)\right)}$$

We use V_g and V_b to denote the green term and the blue term respectively. Because n_g is non-positive, V_g is no smaller than V_b. Consider the following two cases.

- When $V_b \leq 1/2$, the above inequality implies that $\Pr[A]/\Pr[B] \leq 2(N^{-N} + V_g)$.
- When $V_b > 1/2$, $2(N^{-N} + V_g) > 2V_b > 1$. Because $\Pr[A]/\Pr[B]$ corresponds to a conditional probability which is at most 1, $\Pr[A]/\Pr[B] \leq 2(N^{-N} + V_g)$ still holds in this case.

Therefore, we further simplify $\Pr[A]/\Pr[B]$ as the following.

$$\begin{aligned}
\frac{\Pr[A]}{\Pr[B]} &\leq 2\left(N^{-N} + N^{m+n_g} \exp\left(-N^{L-m-n_g}\right)\right) \\
&\leq 2\left(N^{-N} + N^s \exp\left(-N^{L-s}\right)\right), \text{where } s = 1 + lim_s + clim_s^\alpha
\end{aligned}$$

The target theorem can be obtained by applying Theorem 4.11 with the above inequality. □

In Section 5.3, we claim that function Decouling in Algorithm 1 is a probabilistic Occam solver when (1) the error rate of *decoupling* in it is bounded by the Occam-approximation theorem, and (2) client synthesizer $\mathcal{S}_\tau$ is a probabilistic Occam solver. We supplement the details using the following lemma.

Lemma C.11. *Given distribution $\tilde{\Phi}$ over $\varphi_{g,1}$ tasks, define two models $\tilde{\Phi}_{lim}$ and $\tilde{\Phi}_f$ as the following.*

- *$\tilde{\Phi}_{lim}$ represents the induced probability on those considered tasks under scalability threshold lim_s, i.e., $\tilde{\Phi}_{lim}(\varphi') := \Pr_{\varphi\sim\tilde{\Phi}}[\varphi' = \varphi \mid \exists(f_1^*, \tau_1^*) \in \mathbb{P}_{lim_s}, \varphi(f_1^*, \tau_1^*)]$.*
- *$\tilde{\Phi}_f$ represents the distribution on subtask φ_f induced by $\tilde{\Phi}_{lim}$, i.e., $\tilde{\Phi}_f(\varphi') = \Pr_{\varphi\sim\tilde{\Phi}_{lim}}[\varphi' = \varphi_f]$.*

Let $\mathcal{S}_f$, $\mathcal{S}_\tau$ be two client synthesizers satisfying the following conditions.

- *$\mathcal{S}_f$ and decoupling satisfy the condition of the Occam-approximation theorem. In other words, (1) $\mathcal{S}_f$ is an α_1-Occam solver with constant c_1, and (2) the following formula is satisfied.*

$$\forall f_1^* \in \mathbb{P}_{\le lim_s}, \forall f_1 \in \mathcal{S}_f^D(f_1^*)/\{f_1^*\}, \Pr_{\varphi_f\sim\tilde{\Phi}_f}\left[\#\varphi_f(f_1) \mid \#\varphi_f(f_1^*)\right] \le \beta$$

- *$\mathcal{S}_\tau$ is a probabilistic α_2-Occam solver with constant (c_2, γ_2) on any distribution.*

Then Decoupling is a probabilistic α_2-Occam solver with constant $(c_2, 2^{lim_s + c_1 lim_s^{\alpha_1}}\beta + \gamma_2)$ under $\tilde{\Phi}_{lim}$.

Proof. Let $(ms_f(\varphi), ms_\tau(\varphi))$ be the smallest valid solution of task φ, and let $\mathcal{E}$ be the event that $\mathcal{S}_f$ synthesizes ms_φ. When $\mathcal{E}$ happens, let τ_1 be the corresponding combinator synthesized by $\mathcal{S}_\tau$. Because $\mathcal{S}_\tau$ is a probabilistic Occam solver, $size(\tau_1)$ is no larger than $c_2 size(ms_\tau(\varphi))^{\alpha_2}$ with a probability of at least $1-\gamma$. Therefore, Decoupling is a probabilistic α_2-Occam solver with constant $(c_2, \Pr[\neg\mathcal{E}] + \gamma_2)$.

To bound probability $\Pr[\neg\mathcal{E}]$, we use Lemma 4.3. By the proof of the Occam-approximation theorem, the first condition of this lemma implies the condition of Lemma 4.3, i.e.,

$$\forall f_1^* \in \mathbb{P}_{\le lim_s}, \Pr_{\varphi\sim\tilde{\Phi}_{lim}}\left[\mathcal{S}_f(\#\varphi_f) \neq f_1^* \mid \varphi_f(f_1^*)\right] \le 2^{c_1 lim_s^{\alpha_1}}\beta$$

By the definition of $\tilde{\Phi}_{lim}$, for any task φ drawn from this distribution, $ms_f(\varphi)$ must be in $\mathbb{P}_{\le lim_s}$. Using this property, we bound $\Pr[\neg\mathcal{E}]$ via the following derivation.

$$\begin{aligned}
\Pr_{\varphi\sim\tilde{\Phi}_{lim}}[\neg\mathcal{E}] &= \Pr_{\varphi\sim\tilde{\Phi}_{lim}}[\mathcal{S}_f(\#\varphi_f) \neq ms(\varphi)] \\
&\le \sum_{f_1^*\in\mathbb{P}_{lim}} \Pr_{\varphi\sim\tilde{\Phi}_{lim}}[\mathcal{S}_f(\#\varphi_f) \neq f_1^* \wedge ms(\varphi) = f_1^*] \\
&\le \sum_{f_1^*\in\mathbb{P}_{lim}} \Pr_{\varphi\sim\tilde{\Phi}_{lim}}[\mathcal{S}_f(\#\varphi_f) \neq f_1^* \mid \varphi_f(f_1^*)] \\
&\le 2^{lim_s} 2^{c_1 lim_s^{\alpha_1}}\beta = 2^{lim_s + c_1 lim_s^{\alpha_1}}\beta
\end{aligned}$$

Therfore, Decoupling is a probabilistic α_2-Occam solver with constant $(c_2, 2^{lim_s + c_1 lim_s^{\alpha_1}}\beta + \gamma_2)$ under distribution $\tilde{\Phi}_{lim}$. □

D APPENDIX: EVALUATION

D.1 Algorithm Classes Involved in the Evaluation

In this section, we supply some details to the algorithm classes involved in our evaluation. Because the parallel algorithm has been discussed in Section 2, we discuss only the other four classes here. For convenience, we use LSP and RANGE to denote the longest segment problem and the problem of range update and range query respectively.

```
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 = /*τ (A[i], info)*/;
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 }
```

Fig. 4. The sketch of the first algorithm class for LSP.

The first algorithm class for LSP. This algorithm class is for the case where predicate b is both *prefix-closed* and *overlap-closed*, where:

- *prefix-closed* means that $\forall l_1, l_2, b\ (l_1 +\!\!+ l_2) \rightarrow b\ l_1$.
- *overlap-closed* means that $\forall l_1, l_2, l_3, len(l_2) > 0 \wedge b\ (l_1 +\!\!+ l_2) \wedge b\ (l_2 +\!\!+ l_3) \rightarrow b\ (l_1 +\!\!+ l_2 +\!\!+ l_3)$.

The main idea of this algorithm class is to consider all prefixes of the input list l in the increasing order of the size, and calculate the longest suffix satisfying b for each of them. Let $ls(l)$ be the longest suffix of l satisfying predicate b. For two consecutive prefixes l_1 and $l_2 = l_1 +\!\!+ [a]$, when b is both prefix-closed and overlap-closed, $ls(l_2)$ must be one of $ls(l_1) +\!\!+ [a]$, $[a]$ and $[]$, depending on whether $ls(b, l_1) +\!\!+ [a]$ and $[a]$ satisfying b.

Algorithm 4 shows the sketch of this algorithm class. It calculates the longest valid suffix for each prefix of `A`, stores its length as `len` (Line 6), and necessary values on the longest valid suffix as `info` (Line 9). When a new element is considered, `lsp` verifies whether $ls(A[0 \ldots i-1]) +\!\!+ [A_i]$, $[A_i]$, $[]$ are valid, and picks the first valid one among them (Lines 9-18). At this time, operator τ is used to quickly update `info` and verify whether $ls(A[0 \ldots i-1]) +\!\!+ [A_i]$ is valid (Line 9).

To apply this algorithm class, an operator τ and a lifting function f satisfying the following formula are required, where f provides necessary auxiliary values for operator τ.

$$\forall l \in \texttt{List}, \forall a \in \texttt{Int},\ (b \triangle f)\ (l +\!\!+ [a]) = \tau\ \left(a, (b \triangle f)\ l\right)$$

Clearly, the above task is an instance of the lifting problem.

The second algorithm class for LSP. This algorithm class is for the case where predicate b is *prefix-closed*. Similar to the first one, this algorithm class also calculates the longest suffix for each prefix of l. When predicate b is *prefix-closed*, $ls(l_2)$ is always equal to $ls(ls(l_1) +\!\!+ [a])$, where l_1 and $l_2 = l_1 +\!\!+ [a]$ are two consecutive prefixes of l.

```
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 = /*τ1 (A[i], info)*/
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 = /*τ2 (A[i-len], info)*/;
21             len = len - 1;
22         }
23         res = max(res, len);
24     }
25     return res;
26 }
```

Fig. 5. The sketch of the second algorithm class for LSP.

Algorithm 5 shows the sketch of this algorithm class. Similar to the first algorithm class, it also calculates the longest valid suffix for each prefix of A (Lines 8-24). Each time when a new element A_i is considered, this algorithm tries suffixes of $(ls\ (A[0 \ldots i-1]) +\!\!+ [A_i]$ in order (Lines 8-22) and checks whether the current suffix is valid via operator τ_1 (Line 10). If it is not, this algorithm removes the first element via operator τ_2 (Line 20).

To apply this algorithm class, two operators τ_1, τ_2 and a lifting function f satisfying the following formulas are required, where f provides necessary auxiliary values for both operators.

$$\forall l \in \texttt{List}, \forall a \in \texttt{Int},\ (b \triangle f)\ (l +\!\!+ [a]) = \tau_1\ \Big(a, (b \triangle f)\ l\Big)$$

$$\forall l \in \texttt{List}, \forall a \in \texttt{Int},\ (head\ l = a) \rightarrow (b \triangle f)\ (tail\ l) = \tau_2\ \Big(a, (b \triangle f)\ l\Big)$$

The above task can be regarded as an instance of the lifting problem with two operators.

The third algorithm class for LSP. This algorithm class does not have requirement on b. It uses a technique namely *segment partition*. Given list $A[1 \ldots n]$, its segment partition is a series of consecutive segments $(r_0 = 0, r_1], (r_1, r_2], \ldots, (r_{k-1}, r_k = n]$ satisfying (1) $\forall i \in [1, k], \forall j \in (r_{i-1}, r_i), A_j > A_{r_i}$, and (2) $\forall i \in [2, k], A_{r_{i-1}} \leq A_{r_i}$. This algorithm class first generates the segment partition of the given list and then gets the result by merging the information of all lists together.

Figure 6 shows the corresponding sketch. It maintains the segment partition for each prefix of A (Lines 8-15): `num` represents the number of segments in the partition (Line 7), `rpos[i]` represents the index of the right end of the ith segment (Line 5), and `info[i]` records the function values of p

```
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]] > A[i]) {
11             now = /*τ (A[rpos[num]], (info[num], A[rpos[num]]))*/;
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 = /*τ (A[rpos[i]], (info[i], now))*/;
19         merge(info[i], A[rpos[i]], now);
20     }
21     return now.res;
22 }
```

Fig. 6. The sketch of the third algorithm class for LSP.

(i.e., the length of the longest valid segment) and f on the content of the ith segment (Line 1-4). Each time, when a new element is inserted, the algorithm merges the last several segments together using operator τ 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. The algorithm merges these segments together (Lines 16-20) and thus obtains the result (Line 21).

To apply this algorithm class, an operator τ and a lifting function f satisfying the following formula are required, where f provides necessary auxiliary values for operator τ.

$$\forall l_1, l_2 \in \texttt{List}, \forall a \in \texttt{Int}, (\forall a' \in l_1, a' > a \wedge \forall a' \in l_2, a \leq a') \rightarrow \\ (p \triangle f)\ (l_1 +\!\!+ [a] +\!\!+ l_2) = \tau\left(a, (p \triangle f)^2\ (l_1, l_2)\right)$$

Clearly, the above task is an instance of the lifting problem.

The algorithm class for RANGE. This algorithm class uses the segment tree to solve the RANGE task. A segment tree is a tree-like data structure where each vertex corresponds to a segment in the list. On each vertex, several values with respect to the corresponding segment are maintained.

- For each update operator, it distributes the updated range into several disjoint vertices and applies the update on the segment tree structure in batch via *lazy tags*.
- For each query operator, it also distributes the updated range into disjoint vertices and merges the maintained values on these vertices together.

Strictly, this algorithm class requires a constant a_0 and an operator $\otimes$ such that the semantics of the update operator u forms a monoid:

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

Automatically finding a_0 and $\otimes$ is a separate synthesis task. Therefore, in this paper, we assume that a_0 and $\otimes$ are directly given for simplicity.

Figure 7 shows the sketch of this algorithm class. For simplicity, we assume the element in x, the output of the query function h, and the parameter a of the update function u are all integers. This algorithm uses an array `info` to implement the segment tree, where (1) `info[1]` records the information on the root node, which corresponds to the whole list, i.e., range $[0, n-1]$, and (2) `info[k * 2]` and `info[k * 2 + 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 (Lines 1-4). Array `tag` records the lazy tag on each node: `tag[k]` represents that all elements inside the range corresponding to node k should be updated via $\lambda w.u$ (`tag[k]`, w), but such an update has not been considered by all nodes in the subtree of node k yet.

There are several functions used in this algorithm:

- `apply` deals with an update on all elements in the segment corresponding to node `pos` by updating `info[pos]` via operator τ_2 (Line 7).
- `pushdown` applies the tag on node `pos` to its children (Lines 11-12), and clear it (Line 13).
- `initialize` initializes the information for node `pos` which corresponds to range $[l, r]$. It first recurses into two children (Lines 18-19) and then merges the sub-results together via operator τ_1 (Line 20).
- `update` applies an update $([ul, ur], \lambda w.u\ (a, w))$ to node `pos` which corresponds to range $[l, r]$. If $[l, r]$ does not overlap with $[ul, ur]$, the update will be ignored (Line 23). If $[l, r]$ is contained by $[ul, ur]$, the update will be performed via the lazy tag (Line 24). Otherwise, `update` recurses into two children (Lines 26-27) and merges the sub-results together via operator τ_1 (Line 28).
- `query` calculates a subresult for query $[ul, ur]$ by considering elements in node `pos` only. It is implemented in a similar way to function `update`.

To solve a `Range` task, $\mathcal{A}_r$ (1) initializes the segment tree via function `initialize` (Line 37), and then (2) invokes the corresponding functions for each operator (Lines 39-43).

To apply this algorithm class, two operators τ_1, τ_2 and a lifting function f satisfying the following formulas are required, where f provides necessary auxiliary values for both operators.

$$\forall l_1, l_2 \in \texttt{List},\ (h \triangle f)\ (l_1 +\!\!+ l_2) = \tau_1\ \left((q \triangle f)^2\ (l_1, l_2)\right)$$

$$\forall l \in \texttt{List}, \forall a \in \texttt{Int},\ (h \triangle f)\ (map\ (\lambda w, u\ (a, w))\ l) = \tau_2\ \left(a, (h \triangle f)\ l\right)$$

The above task can be regarded as an instance of the lifting problem with two operators.

D.2 Extra Operators

As discussed in Section 7.2, we manually supply operators to *AutoLifter* on 9 tasks in our dataset under the enhanced setting. In this section, we report the details on these extra operators.

Task *atoi* for parallelization. The original program of task *atoi* is shown as Figure 8, which converts a list to an integer by regarding the list as a decimal string.

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

Task *max_sum_between_ones* for parallelization. The original program of this task is shown as Figure 9, which calculates the maximum sum among segments that does not include number 1.

Under the enhanced setting, we supply operator *prefix_till_1* to grammar G_f, where *prefix_till_1* l is defined as the longest prefix of l that does not include 1 as an element.

Task *lis* for parallelization. The original program of task *lis* is shown as Figure 10, which calculates the length of the longest segment such that all elements are ordered.

```
struct Info {
    Int res; // Variable representing h l.
    // Variables representing f l.
}info[N];
Int tag[N];
void apply(int pos, Int a){
    info[pos] = /*τ2 (a, info[pos])*/;
    tag[pos] = tag[pos] ⊗ a;
}
void pushdown(int pos) {
    apply(pos * 2, tag[pos]);
    apply(pos * 2 + 1, tag[pos]);
    tag[pos] = a0;
}
void initialize(int pos, int *A, int l, int r) {
    if (l == r) {info[pos] = /*h [A[l]], f [A[l]]*/; return;}
    int mid = l + r >> 1;
    initialize(pos * 2, A, l, mid);
    initialize(pos * 2 + 1, A, mid + 1, r);
    info[pos] = /*τ1 (info[pos * 2], info[pos * 2 + 1])*/;
}
void update(int pos, int l, int r, int ul, int ur, Int a) {
    if (l > ur || r < ul) return;
    if (l >= ul && r <= ur) {apply(pos, a); return;}
    int mid = l + r >> 1; pushdown(pos);
    update(pos * 2, l, mid, ul, ur, a);
    update(pos * 2 + 1, mid + 1, r, ul, ur, a);
    info[pos] = /*τ1 (info[pos * 2],info[pos * 2 + 1])*/;
}
Info query(int pos, int l, int r, int ul, int ur) {
    if (l > ur || r < ul) return {/*h [], f []*/};
    if (l >= ul && r <= ur) return info[pos];
    int mid = l + r >> 1; pushdown(pos);
    return /*τ1 (query(pos * 2, l, mid, ul, ur), query(pos * 2
         + 1, mid +1, r, ul, ur))*/;
}
void range(int n, int *A, int m, Operator* op) {
    initialize(1, A, 0, n - 1);
    for (int i = 0; i < m; ++i) {
        if (op[i].type == Update) {
            update(1, 0, n - 1, op[i].l, op[i].r, op[i].a);
        } else {
            print(query(1, 0, n - 1, op[i].l, op[i].r));
        }
    }
}
```

Fig. 7. The sketch for the algorithm class of RANGE.

Under the enhanced setting, we supply operator *longest_prefix* to grammar G_f. *longest_prefix* takes a list l and a binary compare operator R as the input, and returns the longest prefix of l that is ordered with respect to R.

```
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  }
```

Fig. 8. The original program of task *atoi*.

```
1  int max_sum_between_1s(int n, int *A) {
2      int ms = 0, cs = 0;
3      for (int i = 0; i < n; ++i) {
4          cs = A[i] != 1 ? cs + A[i] : 0;
5          ms = max(ms, cs);
6      }
7      return ms;
8  }
```

Fig. 9. The original program of task *max_sum_between_ones*.

```
1  int lis(int n, int *A) {
2      int cl = 0, ml = 0, prev = A[0];
3      for (int i = 1; i < n; ++i) {
4          cl = prev < A[i] ? cl + 1 : 0;
5          ml = max(ml, cl);
6          prev = A[i];
7      }
8      return ml;
9  }
```

Fig. 10. The original program of task *lis*.

Task *largest_peak* for parallelization. The original program of task *largest_peak* is shown as Figure 11, which calculates the maximum sum among those segments that all elements are positive.

```
1  int largest_peak(int n, int *A) {
2      int cmo = 0, lpeak = 0;
3      for (int i = 0; i < n; ++i) {
4          cmo = A[i] > 0 ? cmo + A[i] : 0;
5          lpeak = max(cmo, lpeak);
6      }
7      return lpeak;
8  }
```

Fig. 11. The original program of task *largest_peak*.

Under the enhanced setting, we supply operator *longest_prefix*' to grammar G_f. *longest_prefix*' takes a list l and a binary 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* for parallelization. Several tasks in the dataset used by Farzan and Nicolet [2021b] perform regex matching on the list. These tasks can be divided into two categories:

- *longest_reg* asks for the length of the longest segment that matches a given regex.
- *count_reg* asks for the number of segments that matches a given regex.

There are four regex-related 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 tasks *count_1(0*)2*, *longest_1(0*)2*, *longest_odd(0+1)**. Both these operators takes a list l and a regex r as the input, and they return the longest prefix and the longest 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_τ for tasks *longest_(00)**, *longest_odd(0+1)**, because they require the combinator to tell the parity of an integer.

Task *page21* for the longest segment problem. The original program of task *page21* is shown as Figure 12, which calculates the length of the longest segment satisfying that the leftmost element is the minimum and the rightmost element is the maximum.

```
1  int page21(int n, int *A) {
2      int ans = 0;
3      for (int i = 0; i < n; ++i) {
4          int ma = -INF;
5          for (int j = i; j < n; ++j) {
6              if (A[j] < A[i]) break;
7              ma = max(ma, A[j]);
8              if (ma == A[j])
9                  ans = max(ans, j - i + 1);
10         }
11     }
12     return ans;
13 }
```

Fig. 12. The original 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.