Synthesizing Efficient Dynamic Programming Algorithms

Ruyi Ji
Key Lab of High Confidence Software Technologies,
Ministry of Education, School of Computer Science,
Peking University
Beijing, China
jiruyi910387714@pku.edu.cn

Tianran Zhu
Key Lab of High Confidence Software Technologies,
Ministry of Education, School of Computer Science,
Peking University
Beijing, China
moiezen@pku.edu.cn

Yingfei Xiong*
Key Lab of High Confidence Software Technologies,
Ministry of Education, School of Computer Science,
Peking University
Beijing, China
xiongyf@pku.edu.cn

Zhenjiang Hu
Key Lab of High Confidence Software Technologies,
Ministry of Education, School of Computer Science,
Peking University
Beijing, China
huzj@pku.edu.cn

Abstract

Dynamic programming is an important optimization technique, but designing efficient dynamic programming algorithms can be difficult even for professional programmers. Motivated by this point, we propose a synthesizer namely *MetHyl*, which automatically synthesizes efficient dynamic programming algorithms from a possibly inefficient program in the form of *relational hylomorphism*. *MetHyl* consists of a transformation system and efficient synthesis algorithms, where the former transforms a hylomorphism to an efficient dynamic programming algorithm via four synthesis tasks, and the latter solves these tasks efficiently. We evaluate *MetHyl* on 37 tasks related to 16 optimization problems collected from *Introduction to Algorithm*. The results show that *MetHyl* achieves exponential speed-ups on 97.3% problems and the time complexity of the reference solutions on 70.3% problems with an average time cost less than one minute.

1 Introduction

Combinatorial Optimization is a topic on finding an optimal solution from a finite set of valid solutions [Schrijver 2003]. Combinatorial optimization problems (COPs), such as the knapsack problem and the traveling salesman problem, widely exist in various domains. Solving a COP is usually difficult as the number of solutions can be extremely large.

Dynamic programming is an important technique for solving COPs. A dynamic programming algorithm can be implemented in the top-down approach or the bottom-up approach, and the top-down approach is also known as *memoization*. Given a recursive function, memoization can be easily implemented by caching the results of existing calls. Though obtaining an arbitrary memoization algorithm is not difficult, different memoization algorithms could have huge performance differences, and it is challenging to find an *efficient* memoization algorithm for a specific problem.

Motivated by the importance and difficulty of designing memoization, in this paper, we propose a novel approach, *MetHyl*, to **automatically synthesizing efficient memoization for COPs**. This task is challenging because (1) the result program is large and involves complex control flow, and (2) its efficiency needs to be ensured.

To simplify this task, we utilize the theory of *program calculation* [Bird and de Moor 1997]. The theory establishes an algebra for **human users** to derive programs, where various theorems and principles have been developed for manually deriving efficient memoization. In this paper, we use these results to decompose the whole synthesis task into simpler subtasks that are tractable by synthesis techniques.

Concretely, our approach takes a (possibly inefficient) recursive program specified as a (relational) hylomorphism and produces an efficient memoization program. A hylomorphism is a program written in a specific yet general recursive pattern. Meijer et al. [1991] show that every primitive recursive function can be written in a hylomorphism. Different approaches for automatically deriving hylomorphisms have been proposed, including generating a hylomorphism from

*Corresponding author

Authors' addresses: Ruyi Ji, Key Lab of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Tianran Zhu, Key Lab of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, Beijing, China, moiezen@pku.edu.cn; Yingfei Xiong, Key Lab of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Zhenjiang Hu, Key Lab of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, Beijing, China, huzj@pku.edu.cn.

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

a recursive program [Bird and De Moor 1994; Hu et al. 1996] or from a logic specification [Lin et al. 2021].

The first contribution of this paper is a transformation system that transforms a hylomorphism into an efficient memoization through four synthesis subtasks. The transformation is a combination of two existing techniques in program calculation, namely *thinning* [de Moor 1994] and *incrementalization* [Liu and Stoller 2003]. Both techniques were proposed only for a subclass of hylomorphisms, we generalize them to hylomorphisms in our system. The four synthesis subtasks involve only subparts of the program that do not have complex control structures, and thus are tractable.

The second contribution of this paper is efficient synthesis algorithms for solving the four synthesis subtasks. We classify the four subtasks into two categories, and solve them by reducing to a problem that an existing synthesizer can handle, or proposing a new synthesis algorithm for the subtasks.

We implement *MetHyl* and evaluate *MetHyl* on 37 tasks related to 16 COPs collected from *Introduction to Algorithm* [Cormen et al. 2009], a popular textbook for algorithm courses. The results show that *MetHyl* achieves exponential speedups on 97.3% tasks with an average time cost of 59.2 seconds. Moreover, on 70.3% tasks, *MetHyl* achieves the same time complexities as the solutions provided by Cormen et al. [2009] and Li [2011].

2 Overview

In this section, we introduce the main idea of *MetHyl* using a classical COP namely *0/1 knapsack* [Mathews 1896].

> *Given a set of items, each with a weight w_i and a value v_i, put a subset of them in a knapsack of capacity W to get the maximum total value in the knapsack.*

For example, $xs = [(3,3),(2,3),(1,2)], W = 4$ describes an instance of 0/1 knapsack, where three items are available, their weights are 3, 2, 1 respectively, their values are 3, 3, 2 respectively, and the capacity of the knapsack is 4. At this time, the optimal solution is to put the first and the third items, i.e., $[(3,3),(1,2)]$, where the value sum is 5.

To formally describe this COP, we need to specify (1) the set of valid solutions (in this case, subsets of items whose total weight is within the capacity) and (2) the objective value of each solution (in this case, the total value of each subset). In *MetHyl*, the user writes two programs, generator g and scorer o, to specify the two parts. The generator g takes the parameters of the problem (in this case, the list of items) as input and generates all valid solutions for the problem. The scorer o is an objective function that maps each solution to its objective value.

The code below shows the two programs for our example. The parameter g of function g is itself to enable recursion later with a fixed-point combinator and the parameter *is* is the list of items. Functions *sumw* and *sumv* are used to calculate the sum of weights and values for a list of items, respectively. Given a fixed-point combinator *fix*, *fix g* exhaustively returns all sublists, and o calculates the total value of the input list. For simplicity, we shall directly use g to refer to the recursive version, i.e., the first parameter is given.

$$\begin{aligned} g = \lambda g.\lambda xs.\ & \text{if } |xs| = 0 \text{ then } [[]] \\ & \text{else } (g\ (tail\ xs)) +\!\!+ \\ & \quad \Big[(head\ xs) :: p \,\Big|\, p \in g\ (tail\ xs), \\ & \qquad (sumw\ p) + (head\ xs).1 \le W\Big] \\ o = \lambda p.&(sumv\ p) \end{aligned}$$

MetHyl requires g to be a (relational) hylomorphism. The above program demonstrates a simplified version of a hylomorphism on list, and the full definition of hylomorphisms can be found in Section 3. In a nutshell, a hylomorphism on input type A first produces a value from the input (in this case, *head xs*), then recursively applies itself to another value of type A constructed from the input (*tail xs*), and finally combines the results to form the final result. As mentioned before, hylomorphisms could represent a wide range of programs and can be automatically converted from recursive programs or logic specifications.

To distinguish the input and output of the outermost invocation to g and those of other invocations, we call the input of an arbitrary invocation to g a *search state*, as an invocation represents a step in a depth-first search, and call the output of a recursive invocation to g a *partial solution*, as it is not a full solution yet.

We can easily build a memoization algorithm based on the two input programs, as shown below. In the program, m is a function that returns a key for each search state. Two states have the same key only if they lead to the same output. Currently, m is the identity function to trivially ensure this property. Function *mem* implements the memoization algorithm and *buffer* is a global map that stores the result for each key. Finally, *prog* returns the optimal solution from all solutions. Here *argmax o ps* chooses the optimal solution in a list of solutions *ps* based on the objective function o, and *fix* is a fixed-point combinator.

$$\begin{aligned} m\quad &= \lambda xs.xs \\ mem &= \lambda mem.\lambda xs.\ \text{if } buffer[m\ xs] = \bot \text{ then} \\ &\qquad\quad buffer[m\ xs] \leftarrow g\ mem\ xs; \\ &\qquad buffer[m\ xs] \\ prog &= \lambda xs.argmax\ o\ ((fix\ mem)\ xs) \end{aligned}$$

Though the memoization algorithm *prog* has effectively reused the repeated invocations to g, its time complexity is still exponential to the number of items. The goal of our approach is to optimize this inefficient memoization algorithm into an efficient algorithm.

2.1 Transformation System

We first analyze the factors that affect the performance of g, the function consuming most of the execution time. The time complexity of g is as follows.

$$O\left(n_{keys}\left(s_{st}+n_{sols}s_{sol}\right)\right)$$

Here n_{keys} denotes the number of keys that m could possibly return, n_{sols} denotes the number of solutions returned by a recursive call, s_{st} denotes the size of the search state, and s_{sol} denote the size of a partial solution. The execution time of g is a product of the number of invocations and the execution time of a single invocation excluding the recursive call. The former is further confined by n_{keys}. The latter consists of the execution time of processing the input ($|xs|$ takes $O(n)$) and the execution time of producing the solution, where there are n_{sols} solutions, and each takes $O(s_{sol})$ to process. Though this analysis is specific to this input program, the four variables also dominate the performance in general cases.

Our approach reduces the four variables in the above formula, namely, n_{sols}, s_{sol}, n_{keys}, s_{st}. Our goal is to reduce s_{sol} and s_{st} to $O(1)$, and reduce n_{sols} and n_{keys} as much as possible.

Figure 1 shows the process of optimizing our example in *MetHyl*. For simplicity, we omit the first parameter of g. *MetHyl* has four steps, each optimizes one variable.

Step 1. This step optimizes n_{sols}, the number of partial solutions, using the *thinning* method proposed by de Moor [1994]. Thinning uses a preorder R to represent the dominant relationship between partial solutions returned by the recursive invocation to g, where p_1Rp_2 represents that partial solution p_1 can never lead to the global optimal solution if p_2 exists. If such a preorder exists, the result of $(g\ xs)$ can be simplified to $(thin\ R\ (g\ xs))$, where $thin$ is a function returning all minimal values in $(g\ xs)$ in the sense of R.

In program (g_1, o_1, m_1), partial solution p_1 is dominated by p_2 if p_1 not only consumes more capacity but also gains smaller value. For any global solution leaded by p_1, a better solution can be obtained by replacing items in p_1 with p_2. Such a relation can be described by the following preorder.

$$p_1Rp_2 \iff sumw\ p_1 \geq sumw\ p_2 \wedge sumv\ p_1 \leq sumv\ p_2$$

Because the weight sum of partial solutions must be an integer in $[0, W]$, the size of $(thin\ R\ (P\ xs))$ is at most $W+1$. Since normally W and $|xs|$ are in the same order, thinning could reduce the complexity of n_{sols} from exponential to polynomial. Furthermore, for our example, $thin\ R$ could be implemented in linear time, and thus does not increase the time complexity.

As shown in Figure 1, *MetHyl* applies thinning by inserting an invocation of *thin* with an unknown preorder $?R$ to g_1, which is found via program synthesis.

Step 2. This step optimizes s_{sol}, the size of a partial solution, using *incrementalization* proposed by Liu and Stoller [2003]. The goal of incrementalization is to change the representation of a solution from a list to another data structure, so that s_{sol} could be reduced.

To achieve this goal, *MetHyl* identifies all functions in the original program that access a solution, i.e., either take a solution as input or return a solution as output. Then *MetHyl* synthesize new functions to replace them. The new function could take/return a new data type, and thus effectively change the representation.

In this example, *MetHyl* would use a tuple of scalar values, the sum of weights and the sum of values, to replace the list for representing a partial solution. Thus, s_{sol} reduces to $O(1)$.

Note that a COP may require to return a concrete solution. After changing the representation, the optimal solution can be determined by tracing back the calculation that leads to the optimal solution. This is a standard technique, and thus we omit it throughout this paper.

As will be discussed later, our synthesis algorithm is inductive and relies on runtime values collected from the execution of the original program. Since the collected solutions are in the original representation, we synthesize a *converting function* $?f_p$ that converts a solution from the original representation to the new representation.

Step 3. This step optimizes n_{keys}, the number of keys that k possibly returns. *MetHyl* synthesizes a new key function $?f_m$ to replace the original function.

In program (g_3, o_3, m_3), the search state is a list of items, but only the size of the search state matters. *MetHyl* synthesizes $?f_m$ as $\lambda xs.|xs|$. Note that though n_{keys} is not reduced in this example, the synthesized key function still optimizes the program as comparing two keys requires $O(1)$ instead of $O(n)$, where n represents the number of the items.

Step 4. This step optimizes s_{st}, the size of a search state. Similar to Step 2, *MetHyl* uses *incrementalization* to find a new representation for the search state. In this example, *MetHyl* would use the length of the list to represent a search state, reducing s_{st} from $O(n)$ to $O(1)$.

At last, an efficient program (g^*, o^*, m^*) is obtained. Under the assumption that $|xs|$ and W are in the same order, $s_{st} = s_{sol} = O(1)$ and $n_{keys} = n_{sols} = O(n)$ in this program, and thus the total time complexity is $O(n^2)$.

2.2 Correctness Specification Extraction for Program Synthesis

A basic correctness specification is that the new program should be equivalent to the input program. However, such a specification is too complex for program synthesis. To solve this problem, we decompose the correctness requirement for each synthesis task.

Step 1. To ensure the correctness of (g'_1, o'_1, m'_1), *MetHyl* requires $?R$ to satisfy the following two conditions.

1. Let $p \twoheadrightarrow_s p'$ denote a partial solution p' for state s is constructed from another solution p. Let $(g_1\ xs)$, $(g_1\ (tail\ xs))$

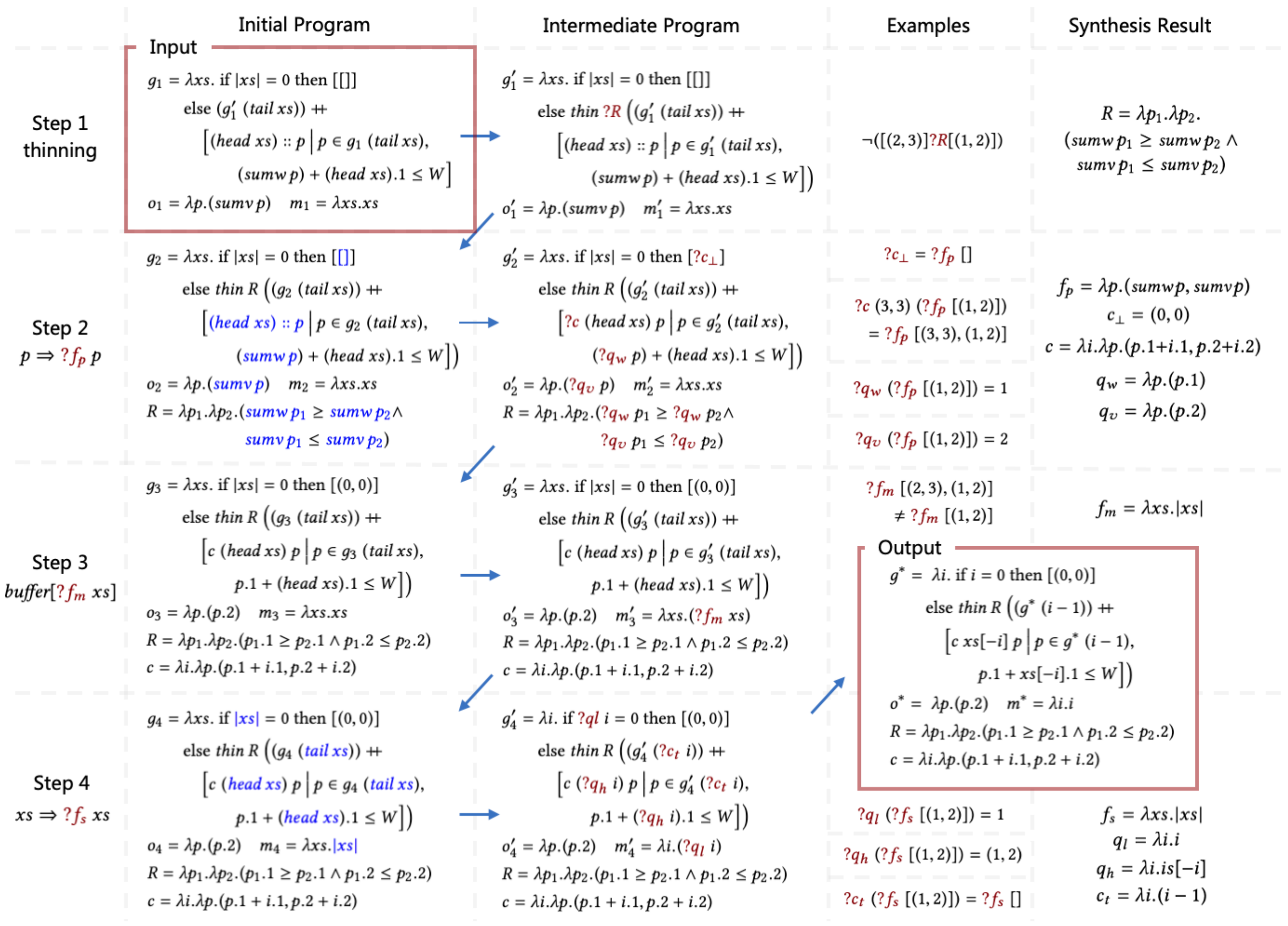
Initial Program
Intermediate Program
Examples
Synthesis Result
Input
Step 1 thinning
$g_1 = \lambda xs.$ if $|xs| = 0$ then [[]] else $(g_1'\ (tail\ xs))$ ++ $[(head\ xs) :: p \mid p \in g_1\ (tail\ xs), (sumw\ p) + (head\ xs).1 \le W]$
$o_1 = \lambda p.(sumv\ p)$ $m_1 = \lambda xs.xs$
$g_1' = \lambda xs.$ if $|xs| = 0$ then [[]] else thin $?R\ ((g_1'\ (tail\ xs))$ ++ $[(head\ xs) :: p \mid p \in g_1'\ (tail\ xs), (sumw\ p) + (head\ xs).1 \le W])$
$o_1' = \lambda p.(sumv\ p)$ $m_1' = \lambda xs.xs$
$\neg([(2,3)]?R[(1,2)])$
$R = \lambda p_1.\lambda p_2.(sumw\ p_1 \ge sumw\ p_2 \wedge sumv\ p_1 \le sumv\ p_2)$
Step 2 $p \Rightarrow ?f_p\ p$
$g_2 = \lambda xs.$ if $|xs| = 0$ then [[]] else thin $R\ ((g_2\ (tail\ xs))$ ++ $[(head\ xs) :: p \mid p \in g_2\ (tail\ xs), (sumw\ p) + (head\ xs).1 \le W])$
$o_2 = \lambda p.(sumv\ p)$ $m_2 = \lambda xs.xs$
$R = \lambda p_1.\lambda p_2.(sumw\ p_1 \ge sumw\ p_2 \wedge sumv\ p_1 \le sumv\ p_2)$
$g_2' = \lambda xs.$ if $|xs| = 0$ then $[?c_\perp]$ else thin $R\ ((g_2'\ (tail\ xs))$ ++ $[?c\ (head\ xs)\ p \mid p \in g_2'\ (tail\ xs), (?q_w\ p) + (head\ xs).1 \le W])$
$o_2' = \lambda p.(?q_v\ p)$ $m_2' = \lambda xs.xs$
$R = \lambda p_1.\lambda p_2.(?q_w\ p_1 \ge ?q_w\ p_2 \wedge ?q_v\ p_1 \le ?q_v\ p_2)$
$?c_\perp = ?f_p\ []$
$?c\ (3,3)\ (?f_p\ [(1,2)]) = ?f_p\ [(3,3),(1,2)]$
$?q_w\ (?f_p\ [(1,2)]) = 1$
$?q_v\ (?f_p\ [(1,2)]) = 2$
$f_p = \lambda p.(sumw\ p, sumv\ p)$
$c_\perp = (0,0)$
$c = \lambda i.\lambda p.(p.1+i.1, p.2+i.2)$
$q_w = \lambda p.(p.1)$
$q_v = \lambda p.(p.2)$
Step 3 buffer$[?f_m\ xs]$
$g_3 = \lambda xs.$ if $|xs| = 0$ then $[(0,0)]$ else thin $R\ ((g_3\ (tail\ xs))$ ++ $[c\ (head\ xs)\ p \mid p \in g_3\ (tail\ xs), p.1 + (head\ xs).1 \le W])$
$o_3 = \lambda p.(p.2)$ $m_3 = \lambda xs.xs$
$R = \lambda p_1.\lambda p_2.(p_1.1 \ge p_2.1 \wedge p_1.2 \le p_2.2)$
$c = \lambda i.\lambda p.(p.1 + i.1, p.2 + i.2)$
$g_3' = \lambda xs.$ if $|xs| = 0$ then $[(0,0)]$ else thin $R\ ((g_3'\ (tail\ xs))$ ++ $[c\ (head\ xs)\ p \mid p \in g_3'\ (tail\ xs), p.1 + (head\ xs).1 \le W])$
$o_3' = \lambda p.(p.2)$ $m_3' = \lambda xs.(?f_m\ xs)$
$R = \lambda p_1.\lambda p_2.(p_1.1 \ge p_2.1 \wedge p_1.2 \le p_2.2)$
$c = \lambda i.\lambda p.(p.1 + i.1, p.2 + i.2)$
$?f_m\ [(2,3),(1,2)] \ne ?f_m\ [(1,2)]$
$f_m = \lambda xs.|xs|$
Output
$g^* = \lambda i.$ if $i = 0$ then $[(0,0)]$ else thin $R\ ((g^*\ (i-1))$ ++ $[c\ xs[-i]\ p \mid p \in g^*\ (i-1), p.1 + xs[-i].1 \le W])$
$o^* = \lambda p.(p.2)$ $m^* = \lambda i.i$
$R = \lambda p_1.\lambda p_2.(p_1.1 \ge p_2.1 \wedge p_1.2 \le p_2.2)$
$c = \lambda i.\lambda p.(p.1 + i.1, p.2 + i.2)$
Step 4 $xs \Rightarrow ?f_s\ xs$
$g_4 = \lambda xs.$ if $|xs| = 0$ then $[(0,0)]$ else thin $R\ ((g_4\ (tail\ xs))$ ++ $[c\ (head\ xs)\ p \mid p \in g_4\ (tail\ xs), p.1 + (head\ xs).1 \le W])$
$o_4 = \lambda p.(p.2)$ $m_4 = \lambda xs.|xs|$
$R = \lambda p_1.\lambda p_2.(p_1.1 \ge p_2.1 \wedge p_1.2 \le p_2.2)$
$c = \lambda i.\lambda p.(p.1 + i.1, p.2 + i.2)$
$g_4' = \lambda i.$ if $?q_l\ i = 0$ then $[(0,0)]$ else thin $R\ ((g_4'\ (?c_t\ i))$ ++ $[c\ (?q_h\ i)\ p \mid p \in g_4'\ (?c_t\ i), p.1 + (?q_h\ i).1 \le W])$
$o_4' = \lambda p.(p.2)$ $m_4' = \lambda i.(?q_l\ i)$
$R = \lambda p_1.\lambda p_2.(p_1.1 \ge p_2.1 \wedge p_1.2 \le p_2.2)$
$c = \lambda i.\lambda p.(p.1 + i.1, p.2 + i.2)$
$?q_l\ (?f_s\ [(1,2)]) = 1$
$?q_h\ (?f_s\ [(1,2)]) = (1,2)$
$?c_t\ (?f_s\ [(1,2)]) = ?f_s\ []$
$f_s = \lambda xs.|xs|$
$q_l = \lambda i.i$
$q_h = \lambda i.is[-i]$
$c_t = \lambda i.(i-1)$

Figure 1. The running procedure for *MetHyl* to synthesize efficient memoization for 0/1 knapsack

be two consecutive invocations, $p_1, p_2 \in g_1\ (tail\ xs)$ and $p_1' \in (g_1\ xs)$ be three solutions returned by the two invocations such that $p_1 \twoheadrightarrow_{xs} p_1'$. The following condition holds.

$$p_1 ?R p_2 \rightarrow \exists p_2' \in (g_1\ xs),\ p_2 \twoheadrightarrow_{xs} p_2' \wedge p_1' ?R p_2'$$

2. For any set of solutions ps, the solution with the largest objective value in ps is always maximal in $?R$.

The first condition ensures that $(g_1'\ xs)$ is always equal to $(thin\ ?R\ (g_1\ xs))$, and the second condition ensures that the solution with the largest objective value is always reserved. Therefore, they together imply the correctness of (g_1', o_1', m_1').

To ensure the second condition, *MetHyl* limits the form of $?R$ to be $\lambda p_1.\lambda p_2.o\ p_1 \le o\ p_2 \wedge p_1 ?R' p_2$, where $?R'$ is another unknown preorder, and then requires only the first condition.

Step 2 and Step 4. In these two steps, the synthesized function should behave the same as the original function. However, since the input and output type of the synthesized function may be different from the original version, we rely on the synthesized converting function for the comparison. Suppose A is the type to be replaced and $?f_p$ is the converting function to be synthesized. For example, given a function q taking A as input, the synthesized function $?q$ should satisfy $?q \circ ?f_p = q$. Similarly, given a function c producing A as output, the synthesized function $?c$ should satisfy $?c = ?f_p \circ q$.

Step 3. To ensure the correctness of (g_3', o_3', m_3'), *MetHyl* requires that $?f_m$ maps two search states is_1 and is_2 to the same key only if $(g_3\ xs_1)$ and $(g_3\ xs_2)$ are exactly the same:

$$?f_m\ xs_1 = ?f_m\ xs_2 \rightarrow g_3\ xs_1 = g_3\ xs_2$$

Our synthesis algorithms use inductive synthesis [Jha and Seshia 2017], and thus we need to collect examples for synthesis. To make the synthesis effective, the examples should reflect a situation that could possibly appear during the program execution. Therefore, we randomly generate inputs to the whole program, and collect runtime values related to the specification.

To enable efficient synthesis for Step 1 and Step 3, we eliminate the logical implication in their specifications and collect simpler examples for $?R$ and $?f_m$.

For Step 1, the fixed form of $?R$ implies the following property.

$$\forall p_1, p_2, (o\ p_1 > o\ p_2) \rightarrow \neg p_1 ?R p_2$$

With the first condition, we obtain the following formula.

$$(\forall p_2' \in (g_1\ xs), \neg p_2 \twoheadrightarrow_{xs} p_2' \vee o\ p_1' > o\ p_2') \rightarrow \neg p_1 ?R p_2$$

The premise of the implication does not contain $?R$, and thus we can collect all pairs of partial solutions (p_1, p_2) where there exists p_1' to satisfy the premise. In this way, we obtain a set of negative examples. Note that when other parts of $?R$ are synthesized, *MetHyl* will collect a new set of examples similarly. More details can be found in Section 5.1.

For Step 3, we transform the original condition to the following equivalent form.

$$g_3\ xs_1 \neq g_3\ xs_2 \rightarrow ?f_m\ xs_1 \neq ?f_m\ xs_2$$

Same as Step 1, the premise of the implication does not contain the synthesized function, and we can collect search state pairs that satisfy the premise.

2.3 Synthesizing Fragments from Examples

The four synthesis tasks can be divided into two categories. **Step 1 and Step 3**. The task in this category is to synthesize a preorder $?R$ satisfying a set of negative examples (a_i, b_i) such that $\neg(a_i ?R b_i)$ holds. The synthesis task for Step 3 can be unified into this form because key function $?f_m$ can be regarded as an equivalence relation for states, which is a special case of preorders.

MetHyl uses an iterative synthesis algorithm to solve this task. To optimize the efficiency of the synthesis algorithm, *MetHyl* synthesizes $?R$ in the following form, which is a conjunction of comparisons on several key functions.

$$a\ R\ b \iff \wedge_i (?key_i\ a)\ ?op_i\ (?key_i\ b)$$

where $?op_i$ is in $\{\leq, =\}$ and $\{=\}$ in Step 1 and 3 respectively.

In this form, $?R$ satisfies a negative example (a_i, b_i) if and only if at least one comparison is evaluated to be false on (a_i, b_i). Therefore, for any preorder that satisfies all examples with n_c comparisons, there must be a comparison that is evaluated to false on at least $1/n_c$ portion of examples.

Motivated by this property, *MetHyl* synthesizes with a limit n_c on the number of comparisons. While determining the ith comparison, only those that are evaluated to false on at least $1/(n_c - i + 1)$ portion of examples are considered. In this way, the search space of preorders is greatly reduced.

Besides satisfying the correctness specification, the other goal of synthesis is to minimize n_{sols} and n_{keys}. We design two objective functions to estimate the effects of the synthesized preorder on minimizing the two variables and then optimize the objective functions via *branch-and-bound*, a commonly used method to solve synthesis problems with an optimization goal [Ji et al. 2020; Lee et al. 2018].

Step 2 and Step 4. In this task, we need to synthesize a series of functions, where each function is independently specified except for the converting function $?f$, which is shared in all specifications. These specifications follow the same form, which requires the composition of an unknown function and $?f$ to be equal to some known function or the composition of $?f$ and some known function.

This task is a hybrid instance of *lifting problem* and *partial lifting problem*. We have proposed an efficient solver, *AutoLifter* [Ji et al. 2021b], for both problems in our previous work. In *MetHyl*, *AutoLifter* is applied to solve this hybrid task. There are some details remaining on merging the results of partial lifting problems and listing problems, which are left to Section 5.2.

AutoLifter uses grammars to guarantee the efficiency of the synthesis results. Under its default setting, $?f$ is synthesized from a grammar including only polynomial-time programs that output only scalar values, and other functions are synthesized from a grammar including only constant-time operators. In this way, the time complexities of all synthesis results except $?f$ are guaranteed to be $O(1)$. In our transformation system, because $?f$ is a converting function and is invoked only once to process the instance, such a guarantee already ensures the efficiency of the transformation result.

3 Preliminaries

To operate functions, the following four operators $\circ, +, \times$, and $\triangle$ will be used in the remainder of this paper.

$$(f_1 \circ f_2)\ x := f_1\ (f_2\ x) \quad (f_1 + f_2)\ (i, x) := f_i\ x, i \in \{0, 1\}$$
$$(f_1 \times f_2)\ (x, y) := (f_1\ x, f_2\ y) \quad (f_1 \triangle f_2)\ x := (f_1\ x, f_2\ x)$$

3.1 Categorical Functors

Functor is an important concept in category theory. A category consists of a set of objects, denoted by uppercase letters such as A, and a set of arrows between objects, denoted by lowercase letters such as f. In this paper, we focus on category **Fun**, where an object is a type and an arrow from object A to object B is a total function from A to B.

In a category, a functor F maps objects to objects, arrows to arrows, and keeps both identity and composition.

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

where id_A represents the identity function on set A. Intuitively, a functor can be regarded as a higher-order function, which constructs new functions from existing functions.

A functor is a *polynomial functor* if it is constructed by identity functor I, constant functors $!A$, and bifunctors $\times, +$. Their definitions are shown below.

$$\mathsf{I}A := A \quad \mathsf{I}f := f \quad (!A)B := A \quad (!A)f := id_A$$
$$(\mathsf{F}_1 \times \mathsf{F}_2)A := \mathsf{F}_1 A \times \mathsf{F}_2 A \quad (\mathsf{F}_1 \times \mathsf{F}_2)f := \mathsf{F}_1 f \times \mathsf{F}_2 f$$
$$(\mathsf{F}_1 + \mathsf{F}_2)A := (\{1\} \times \mathsf{F}_1 A) \cup (\{2\} \times \mathsf{F}_2 A)$$
$$(\mathsf{F}_1 + \mathsf{F}_2)f := \mathsf{F}_1 f + \mathsf{F}_2 f$$

where $A \times B$ is the Cartesian product of objects A and B.

In this paper, when using symbol F, we inherently assume that F is a polynomial functor. Besides, we also use the power functor P to express operations related to power sets.

$$\mathsf{P}A \coloneqq \{s \mid s \subseteq A\} \quad \mathsf{P}f\ s \coloneqq \{f\ a \mid a \in s\}$$

3.2 Generator, Memoization, and Hylomorphism

The concept of *relational hylomorphism* originally defined on another category namely **Rel**. For simplicity, in this paper, we introduce it as its simplified counterpart in category **Fun**.

Definition 3.1 (Recursive Generator)**.** Given arrows $\phi : \mathsf{PF}A \to \mathsf{P}A$ and $\psi : B \to \mathsf{PF}B$, recursive generator $rg(\phi,\psi)_\mathsf{F} : A \to \mathsf{P}B$ is the smallest solution of the following equation, where arrow r_1 is smaller than r_2 if $\forall i, r_1\ i \subseteq r_2\ i$.

$$rg(\phi,\psi)_\mathsf{F} = \phi \circ cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}rg(\phi,\psi)_\mathsf{F}) \circ \psi$$

In this equation, $car[\mathsf{F}] : \mathsf{FP}A \to \mathsf{PF}A$ is defined as the following, which operates similarly with the Cartesian product, and $cup : \mathsf{PP}A \to \mathsf{P}A$ is defined as $cup\ x \coloneqq \cup_{s\in x} s$, which returns the union of all sets in a set of sets.

$$car[\mathsf{I}]\ x \coloneqq x \quad car[\mathsf{F}_1 \times \mathsf{F}_2]\ (x_1,x_2) \coloneqq (car[\mathsf{F}_1]\ x_1) \times (car[\mathsf{F}_2]\ x_2)$$
$$car[!A]\ x \coloneqq x \quad car[\mathsf{F}_1 + \mathsf{F}_2]\ (i,x) \coloneqq \{(i,a) \mid a \in x\}$$

The concept of recursive generators corresponds to recursive programs in **Rel**. As discussed in Section 2, a recursive generator can be naturally memoized via a key function m. We use r^m to denote the result of memoization.

Definition 3.2 (Relational Hylomorphism)**.** Given two arrows $\phi : \mathsf{F}A \to \mathsf{P}A$ and $\psi : B \to \mathsf{PF}B$, relational hylomorphism $[\![\phi,\psi]\!]_\mathsf{F} : B \to \mathsf{P}A$ is defined as $rg(cup \circ \mathsf{P}\phi, \psi)$.

Comparing with general recursive generators, hylomorphism assumes the independence while constructing new solutions. Given the set including all results of the recursion, i.e., $(cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}rg(\phi,\psi)_\mathsf{F}) \circ \psi)$, a hylomorphism independently generates solutions for each result via $\mathsf{P}\phi$, and then merges all solutions via *cup*.

Figure 2 shows a program *prog* corresponding to the input program (g_1, o_1) discussed in Section 2, where g_1 is expressed by relational hylomorphism $[\![\phi,\psi]\!]_\mathsf{F}$. Because both functions ϕ, ψ return a set, we use $|collect|$ instead of $|return|$ to express their outputs for convenience, where each execution of $|collect\ e|$ inserts the value of e to the returning set.

Relational hylomorphisms are natural for specifying COPs, where ψ and ϕ specify the decision procedure and the construction of solutions respectively. However, relational hylomorphism cannot express optimizations (e.g., thinning and branch-and-bound), because it simply returns the union of all constructed solutions. Therefore, the same as the theory of program calculation, *MetHyl* takes a relational hylomorphism as the input but expresses the internal optimized programs via recursive generators.

3.3 Programs

In *MetHyl*, a program is represented by a pair (g, o). Given an instance i, the output of (g, o) is equal to $argmax\ o\ (g\ i)$. In terms of solving COPs, two programs are equivalent if they achieve the same objective value.

Definition 3.3 (Equivalence)**.** Two programs (P_1, S_1) and (P_2, S_2) are equivalence on instance i, denoted as $(P_1, S_1) \sim_i (P_2, S_2)$, if $\max(S_1\ p), p \in (P_1\ i) = \max(S_2\ p), p \in (P_2\ i)$.

In this paper, we provide a simple language $\mathcal{L}_H$ for specifying COPs via relational hylomorphisms. The syntax of this language is shown as Figure 4. To show its usage, Figure 2, 5 and 3 show three different programs in $\mathcal{L}_H$ for describing *0/1 knapsack*, where $|if\ \mathbb{E}\ then\ \mathbb{S}\ else\ \mathbb{S}|$ is a sugar of $|if\ \mathbb{E}\ then\ \mathbb{S}\ else\ skip;|$, and $|\lambda(x_1,x_2).\mathbb{S}|$ is a sugar of extracting components in the input for $|\lambda x.\mathbb{S}|$.

3.4 Thinning

The definition of thinning is based on *preorders*. A preorder on object A is a relation that is reflexive and transitive.

$$\forall a \in A, aRa \quad \forall a,b,c \in A, aRb \wedge bRc \to aRc$$

Definition 3.4. Given a preorder R on A, *thin* $R : \mathsf{P}A \to \mathsf{P}A$ is an arrow such that for any set $s \subseteq A$, *thin* $R\ s$ is the smallest subset of s satisfying $\forall a \in s, \exists b \in thin\ R\ s, aRb$.

One noticeable special case of *thin* is when R is the conjunction of several comparisons on several key functions.

Definition 3.5 (Keyword Preorder)**.** A keyword preorder R of comparisons $\{(op_i, k_i)\}$ on A is defined as the following.

$$aRb \iff \wedge_i (k_i\ a)\ op_i (k_i\ b)$$

where k_i is an arrow from A to `Int` and $op_i \in \{\leq, =\}$.

Theorem 3.6. *Given a keyword preorder R of $\{(op_i, k_i)\}$, define function $N_R(S)$ as the following, where $range(k, S)$ is the range of k on S, i.e., $\max_{a\in S}(k\ a) - \min_{a\in S}(k\ a)$, and $\max(S)$ returns the largest element in S with default value* 1.

$$N_R(S) \coloneqq \left(\prod_i range(k_i, S)\right) \Big/ \max_i \left(range(k_i, S)\ \Big|\ op_i \in \{\leq\}\right)$$

- *For any set S, $|thin\ R\ S| \leq N_R(S)$.*
- *There is an implementation of $(thin\ R)$ with time complexity $O(N_R(S) size(R) + T_R(S))$, where S is the input set, $size(R)$ is the number of comparisons in R, $T_R(s)$ is the time complexity of evaluating all key functions in R for all elements in S.*

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

3.5 Lifting and Partial Lifting

Lifting problems and partial lifting problems are synthsis tasks we studied in our previous work [Ji et al. 2021b]. They are generalized from the task of parallelization.

```
$prog = ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$, where
  $\mathsf{F} = !\mathtt{Unit} + \mathsf{I} + ((!\mathtt{Int} \times !\mathtt{Int}) \times \mathsf{I})$
  $\psi = \lambda xs.$ if $|xs| = 0$ then collect $(1, unit)$;
    else {collect $(2, tail\ xs)$;
      collect $(3, (head\ xs, tail\ xs))$; }
  $\phi = \lambda(tag, p).$ if $tag = 1$ then collect $[]$;
    else if $tag = 2$ then collect $p.2$;
    else if $(sumw\ p.2) + p.1.1 \leq W$ then
      collect $(p.1 :: p.2)$;
  $o = \lambda p.sumv\ p$
```

Figure 2. The program corresponding to (g_1, o_1).

```
$prog_2 = ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$, where
  $\mathsf{F} = !\mathtt{Unit} + ((!\mathtt{Int} \times !\mathtt{Int}) \times \mathsf{I})$
  $\psi = \lambda xs.$ if $|xs| = 0$ then collect $(1, unit)$;
    else collect $(2, (head\ xs, tail\ xs))$;
  $\phi = \lambda(tag, p).$ if $tag = 1$ then collect $[]$;
    else {collect $(p.1 :: p.2)$; collect $p.2$; }
  $o = \lambda p.(sumw\ p \leq W)\ ?\ (sumv\ p) : -\infty$;
```

Figure 3. A valid program for 0/1 knapsack in $\mathcal{L}_H$. In this program, $-\infty$ is a small enough integer that is used to exclude invalid programs.

Program	$\mathbb{P}$	$\rightarrow$	$(\mathbb{H}, \mathbb{E})$
Hylomorphism	$\mathbb{H}$	$\rightarrow$	$[\![\lambda x.\mathbb{S}, \lambda x.\mathbb{S}]\!]_{\mathbb{F}}$
Functor	$\mathbb{F}$	$\rightarrow$	$\mathsf{I} \mid !\mathtt{Unit} \mid !\mathtt{Int} \mid \mathbb{F} \times \mathbb{F} \mid \mathbb{F} + \mathbb{F}$
Statement	$\mathbb{S}$	$\rightarrow$	skip; $\mid \mathbb{S}\ \mathbb{S} \mid$ if $\mathbb{E}$ then $\mathbb{S}$ else $\mathbb{S}$
		$\mid$	collect $\mathbb{E}$; $\mid$ foreach $x \in \mathbb{E}$ in $\mathbb{S}$
Expression	$\mathbb{E}$	$\rightarrow$	$x \mid$ const $\mid \lambda x.\mathbb{E} \mid \oplus\ \mathbb{E} \dots \mathbb{E}$

Figure 4. The syntax of language $\mathcal{L}_H$, where x represents a variable, and $\oplus$ represents a black-box operator.

```
$prog_3 = ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$, where
  $\mathsf{F} = !\mathtt{Unit} + \mathsf{I} + ((!\mathtt{Int} \times !\mathtt{Int}) \times \mathsf{I})$
  $\psi = \lambda(xs, p).$ if $|xs| = 0$ then collect $(1, unit)$;
    else {collect $(2, (tail\ xs, p))$;
      if $(sumw\ p) + (head\ xs).1 \leq W$ then
        collect $(3, (head\ xs, (tail\ xs, (head\ xs) :: p)))$; }
  $\phi = \lambda(tag, p).$if $tag = 1$ then collect $[]$;
    else if $tag = 2$ then collect $p$; else collect $(p.1 :: p.2)$;
  $o = \lambda p.sumv\ p$
```

Figure 5. A valid program for 0/1 knapsack in $\mathcal{L}_H$. The search state in this program is a pair (xs, p), where xs is the list of remaining items, p is the list of selected items.

Definition 3.7. Given arrows p, h starting from object A, a set M including n arrows $m_i : \mathsf{F}_{m_i} A \rightarrow A$, and example space E attaching a set of examples $E[m_i]$ to each $m_i \in M$, lifting problem $\mathsf{LP}(M, p, h, E)$ and partial lifting problem $\mathsf{PLP}(M, p, h, E)$ are to find $?f, ?c_1, \dots, ?c_n$ such that Equation 1 and 2 are satisfied for all $m_i \in M$, respectively.

$$\forall e \in E[m_i], ((p \triangle ?f) \circ m_i)\ e = (?c_i \circ \mathsf{F}_{m_i}(h \triangle ?f))\ e \qquad (1)$$

$$\forall e \in E[m_i], (p \circ m_i)\ e = (?c_i \circ \mathsf{F}_{m_i}(h \triangle ?f))\ e \qquad (2)$$

AutoLifter [Ji et al. 2021b] is an efficient synthesizer for these two tasks, and guarantees that the time complexities of $?f$ and $?c$ are polynomial-time and constant-time respectively.

4 Transformation System

Given a recursive generator r and a search state s, invoking r on s without memoization generates a search tree where each vertex corresponds to a search state. We use two notations T_r and S_r to access the structure of this tree, where $T_r\ s$ and $S_r\ s$ represent the set including all direct children of s and the set including all states in the subtree of s respectively.

For simplicity, in this section we assume that each transition, i.e., each element in the output of ϕ, involves at most one search state. The discussion in this section can be easily extended to the general case, which can be found in Appendix A.2.

4.1 Step 1: Thinning

Given program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and a set of instances I, *MetHyl* returns a program in the following form while keeping the correctness on set I, where $?R$ is a keyword preorder.

$$(rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o) \qquad (3)$$

First, Lemma 4.1 provides a sufficient condition for $?R$ to guarantee the correctness of the result. Recall that notation $p \twoheadrightarrow_s p'$ is introduced in Section 2 to represent that partial solution p' in $(h\ s)$ is constructed from partial solution p.

Lemma 4.1. *Given instance i and program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$, $(rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o) \sim_i ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$ for keyword preorder $?R$ if $?R$ satisfies that (1) $(\leq, o) \in\ ?R$, and (2) for each pair (p_1, p_2) of partial solutions generated from the same invocation of h, $p_1 ?R p_2$ implies that p_1 is dominated by p_2 in the sense of R when constructing new solutions, i.e.,*

$$\forall s \in (S_h\ i), \forall s' \in (T_h\ s), \forall p_1, p_2 \in (h\ s'), p_1 ?R p_2 \rightarrow$$
$$\forall p_1', \left(p_1 \twoheadrightarrow_s p_1' \rightarrow \exists p_2', p_2 \twoheadrightarrow_s p_2' \wedge p_1' ?R p_2'\right) \qquad (4)$$

Given a preorder R and an instance i, a set of counterexamples $CE(R, i)$ can be extracted via Lemma 4.1, which includes all pairs of partial solutions (p_1, p_2) that violates Equation 4. Note that $CE(R, i)$ changes with the choice of R.

Lemma 4.2 shows that to find a correct keyword preorder R_2 by enlarging a keyword preorder R_1, all counterexamples of R_1 must be rejected by new comparisons in R_2.

Lemma 4.2. *Given instance i, for any two keyword preorders $R_1 \subseteq R_2$, the following formula is always satisfied.*

$$\forall (p_1, p_2) \in CE(R_1, i), (p_1, p_2) \notin CE(R_2, i) \leftrightarrow \neg p_1 (R_2/R_1) p_2$$

where R_2/R_1 represents the keyword preorder formed by the new comparisons in R_2 compared with R_1.

Second, for efficiency of the result, the number of plans returned by $(thin\ ?R)$ and its time cost should be minimized. By Theorem 3.6, *MetHyl* defines this cost as the following.

$$cost(?R, I) := N_{?R}(P), \quad P = \left\{ p \middle| i \in I, s \in S_h\ i, p \in h\ s \right\}$$

where P represents all partial solutions related to set I.

In summary, the task for Step 1 is to synthesize a keyword preorder $?R$ satisfying the following requirement.

- (Correctness) $(\leq, o) \in R$ and $\forall i \in I, CE(?R, i) = \emptyset$.
- (Efficiency) $cost(?R, I)$ is minimized.

4.2 Step 2: Incrementalization for Plans

The initial program for Step 2 is in Form 3, where the body of ϕ and ψ are statements ($\mathbb{S}$) in language $\mathcal{L}_H$ (Figure 4).

MetHyl performs Step 2 only when the partial solutions generated by ϕ includes recursive data structures such as lists and trees. Concretely, *MetHyl* constructs a program $prog_2'$ that performs almost the same with $prog_2$ except each partial solution p in $prog_2$ is stored as $?f_p\ p$, where the output of $?f_p$ is limited to a tuple of scalar values. The construction rewrites two types of solution-related functions in $prog_2$.

- Queries, which returns a tuple of scalar values.
- Constructors, which constructs a new partial solution.

The following shows the content of $prog_2'$.

$$prog_2' = (rg((thin\ R') \circ cup \circ \mathsf{P}\phi', \psi)_{\mathsf{F}}, ?q[o]) \qquad (5)$$

- Both o and keywords in $?R$ are queries. In $prog_2'$, o and R are replaced with $?q[o]$ and $R' = \{(op, ?q[k]) \mid (op, k) \in R\}$.
- Constructors and other queries are extracted from ϕ. In $prog_2'$, ϕ is replaced with function ϕ' constructed from ϕ.

ϕ' is constructed via a structural recursion on the AST of ϕ. This procedure is shown as Algorithm 1.

- $|s|$ corresponds to the input variable of ϕ.
- `RewriteE` and `RewriteS` corresponds to non-terminal $\mathbb{E}$ and $\mathbb{S}$ in $\mathcal{L}_H$ respectively. Specially, `RewriteE` returns $\perp$ if the current expression in $prog_2$ is inside a query.
- `Children`$(p, \mathbb{N})$ returns all children of AST node p corresponding to non-terminal $\mathbb{N}$, and `Clone`(p, c) constructs a new AST node by replaces the children of p with list c.

To characterize the specification of $?f_p$, $?q$ and $?c$, we introduce two notations $\mathsf{F}[p]$ and $E(p, i)$.

Algorithm 1: Construct ϕ' from ϕ.

```
Input: A program $\phi = \lambda s.\phi_s$, where $p_s$ is a statement in $\mathcal{L}_H$.
Output: Queries $Q$, Constructors $M$, and $\phi'$ in $prog_2$.
$Q \leftarrow \emptyset$;  $M \leftarrow \emptyset$;
Function RewriteE($p_e$):
    if $|s| \notin p_e$ then return $p_e$;
    if $p_e = |s|$ then return $\perp$;
    $sp_e' \leftarrow \{$RewriteE($sp_e$) $\mid sp_e \in$ Children($p_s, \mathbb{E}$)$\}$;
    if $\perp \in sp_e'$ then
        if $p_e$ output a tuple of scalar values then
            $t_1, \dots, t_m \leftarrow$ temporary variables in $p_e$;
            $Q$.Insert($p_e$); return $|?q[p_e]\ (s, t_1, \dots, t_m)|$;
        end
        return $\perp$;
    end
    return Clone($p_e, sp_e'$);
Function RewriteS($p_s$):
    if $p_2 = |collect\ sp_e|$ then
        $t_1, \dots, t_n \leftarrow$ temporary variables in $p_s$;
        $M$.Insert($sp_e$);  return $|?c[p_s]\ (s, t_1, \dots, t_n)|$;
    end
    $sp_e' \leftarrow [$RewriteE($sp_e$) $\mid sp_e \in$ Children($p_s, \mathbb{E}$)$]$;
    $sp_s' \leftarrow [$RewriteS($sp_s$) $\mid sp_e \in$ Children($p_s, \mathbb{S}$)$]$;
    return Clone($p_s, sp_e' +\!\!+ sp_s'$).
$\phi_s' \leftarrow$ RewriteS($\phi_s$); return $Q, M, \lambda s.\phi_s'$;
```

- For each query (or constructor) p, functor $\mathsf{F}[p]$ indicates partial solutions in the input of p. For queries extracted from R and S, $\mathsf{F}[p] := \mathsf{I}$; For functions extracted from ϕ, $\mathsf{F}[p] := \mathsf{F} \times !T_1 \times \dots !T_n$, where F is the functor in $prog_2$ and $!T_i$ is the type of the ith temporary variable used by p.
- Given instance i, for each query (or constructor) p, set $RE(p, i)$ records the inputs on which p is invoked during $(prog_2\ i)$, which can be obtained by instrumenting $prog_2$.

Lemma 4.3 provides a sufficient condition for $?f_p$, $?q$ and $?c$ to guarantee the correctness of $prog_2'$.

Lemma 4.3. *Given instance i and program $prog_2$ in Form 3, let $prog_2'$ be the program constructed by MetHyl in Step 2. $prog_2 \sim_i prog_2'$ if for any query q and any constructor m, Formula 6 and Formula 7 are satisfied respectively.*

$$\forall e \in RE(q, i), q\ e = ?q[q]\ (\mathsf{F}[q] ?f_p\ e) \qquad (6)$$

$$\forall e \in RE(m, i), ?f_p\ (m\ e) = ?c[m]\ (\mathsf{F}[m] ?f_p\ e) \qquad (7)$$

In summary, the synthesis task for Step 2 is to find $?f_p$, $?q$ and $?c$ satisfying the following two requirements.

- (Correctness) Lemma 4.3 is satisfied for any $i \in I$.
- (Efficiency) The output of $?f_p$ is a tuple of scalar values.

4.3 Step 3: Memoization

In Step 3, given program (r, o), where r is a recursive generator, and a set of instances I, *MetHyl* returns $(r^{?f_m}, o)$, i.e., memoizes r, while keeping the correctness on set I.

For the correctness, Lemma 4.4 provides a sufficient condition for $?f_m$ to guarantee the correctness of $(r^{?f_m}, o)$.

Lemma 4.4. *Given instance i and program (r, o), where r is a recursive generator, $(r^{?f_m}, o) \sim_i (r, o)$ if for any two states $s_1, s_2 \in (S_r\ i)$, $(?f_m\ s_1 = ?f_m\ s_2) \to (r\ s_1 = r\ s_2)$.*

Given an instance i, a set of examples $EM(i)$ can be extracted according to Lemma 4.4. Each example in $EM(i)$ is a set of states (s_1, s_2), representing that $?f_m$ should output differently on these two states.

For efficiency, the number of stored results, which is the number of different outputs produced by $?f_m$, should be minimized. This objective leads to a cost function for $?f_m$.

$$cost(?f_m, I) := \left| \left\{ ?f_m\ s \,\middle|\, i \in I, s \in (S_r\ i) \right\} \right|$$

In summary, the synthesis task of Step 3 is to find $?f_m$ satisfying the following two requirements.

- (Correctness) $\forall i \in I, \forall (s_1, s_2) \in EM(i), ?f_m\ s_1 \neq ?f_m\ s_2$.
- (Efficiency) $cost(?f_m, I)$ is minimized.

4.4 Step 4: Incrementalization for States

After Step 3, the program must be in the following form.

$$prog_4 = (r, o), \quad r = rg\Big((thin\ R) \circ cup \circ \mathsf{P}\phi, \psi\Big)_{\mathsf{F}}^{f_m}$$

where the body of ϕ and ψ are statements ($\mathbb{S}$) in $\mathcal{L}_H$.

MetHyl performs Step 4 only when the type of states includes recursive data structures. Similar to Step 2, *MetHyl* rewrites all state-related queries and constructors in $prog_4$ with $?q$ and $?c$ respectively, and thus constructs a program $prog_4'$ that performs almost the same with $prog_4$ except each state s in $prog_4$ is stored as $?f_s\ s$. To guarantee the efficiency, the output of $?f_s$ is limited to a tuple of scalar values.

In $prog_4$, key function f_m is identified as a query. Other queries and constructors are extracted from ψ similarly with Algorithm 1. The only difference is that in ψ, the parameter sp_e of $|collect\ sp_e|$ is a structure including substates, on which r will be recursively applied, and decision values, which will be directly passed to ϕ. Because F is a polynomial functor, these components can be extracted from sp_e via the access operator, i.e., $|sp_e.i_1.i_2 \dots i_n|$. Those components corresponding to substates and decision values are identified as constructors and queries respectively.

We omit the specification of $?f_s, ?q, ?c$ and the synthesis tasks here because they are almost the same as Step 2.

5 Synthesizer

5.1 Preorder Synthesis

For the synthesis task of Steps 1 and 3, we start with a subproblem where a finite set of comparisons C and a size limit n_c are provided. In this subproblem, the search space of $?R$ is fixed to preorders constructed by $(\leq, o)$, which is necessary according to Theorem 4.1, and at most n_c comparisons in C.

Algorithm 2: Synthesizing preorder $?R$ for Step 1.

```
Input: Input program $(h, o)$ for Step 1, a set of instances $I$, a
       set of comparisons $C = \{(op_i, k_i)\}$, and a size limit $n_c$.
Output: A keyword preorder $?R \subseteq C$ for $(h, S)$.
1  Function BestPreorder($R, lim, costLim$):
2      $es \leftarrow \cup_{i \in I} CE(R, i)$;
3      if $es = \emptyset$ then return $R$;
4      if $lim = 0$ then return $\top$;
5      $cList \leftarrow$ CandidateComps($C, lim, es$);
6      Sort $cList$ in the increasing order of $cost(R \cup \{c\}, I)$;
7      $res \leftarrow \perp$;
8      foreach $c \in cList$ do
9          if $cost(R \cup \{c\}, I) \geq costLim$ then continue;
10         $R' \leftarrow$ BestPreorder($R \cup \{c\}, lim - 1, costLim$);
11         if $R' \neq \perp$ then $(res, costLim) \leftarrow R', (cost(R', I))$;
12     end
13     return $res$;
14 return BestPreorder($\{(\leq, o)\}, n_c, +\infty$);
```

As shown in Algorithm 2, *MetHyl* solves this subproblem via *branch-and-bound*. BestPreorder decides comparisons used in $?R$ one by one with an upper bound on the cost (Lines 1-14). In each turn, a set of candidate comparisons is identified via CandidateComps (Line 5) and they are considered in the increasing order of the size (Line 6). For each comparison c, if its cost is smaller than the bound (Line 9), preorders including c will be considered recursively (Line 10). Results of recursions will be used to update the bound (Line 11).

According to Lemma 5.1, CandidateComps(C, lim, es) returns comparisons that satisfy at least $|es|/lim$ examples.

Lemma 5.1. *Given a set of instances I, for any two keyword preorders $R_1 \subseteq R_2$ where $\forall i \in I, CE(R_2, i) = \emptyset$, there exists a comparison $(op, k) \in R_2/R_1$ satisfying at least $1/(|R_2| - |R_1|)$ portion of examples in $CE(R_1, I) = \cup_{i \in I} CE(R_1, i)$, i.e.,*

$$\left| \left\{ (p_1, p_2) \in CE(R_1, I) \,\middle|\, \neg\Big((k\ p_1) op (k\ p_2)\Big) \right\} \right| \geq \frac{|CE(R_1, I)|}{|R_2| - |R_1|}$$

There are several details remaining to make Algorithm 2 useful in *MetHyl*. We describe these details below.

Decide a finite set of comparisons. In practice, the set of comparisons is usually provided by a grammar, in which the number of comparisons is infinite. Due to the difficulty of finding the optimal preorder from an infinite set of comparisons, *MetHyl* approximates it via the principle of *Occam's Razor*. Concretely, *MetHyl* selects a parameter s_c and takes the smallest s_c comparisons in the grammar as set C.

Decide s_c and n_c. In practice, both s_c and n_c are not given. *MetHyl* decides them iteratively with two parameters s_c^* and n_c^*. In each turn, *MetHyl* considers the subproblem where $n_c = n_c^*$ and $s_c = s_c^*$. If there is no solution, s_c^* will be doubled and n_c^* will be increased by 1 in the next iteration.

Accelerate `CandidateComps`. A direct implementation of `CandidateComps`(C, lim, es) in Algorithm 2 (Line 5) is to evaluate all comparisons in C on all examples in es. Such an implementation is inefficient because both C and es can be large. *MetHyl* improves this point by sampling.

In *MetHyl*, a comparison will be evaluated on all examples only when it satisfies $k_t = \lfloor lim/2 \rfloor$ examples among $lim \times n_t$ random examples, where n_t is a parameter of *MetHyl*. By Chernoff bound, the probability for a comparison that satisfies at least $1/lim$ portion of examples in es to fail on this condition is at most $\exp(-n_t/8)$. Specially, k_t is set to $lim \times n_t$, i.e., the number of samples, when $lim = 1$.

Note that though the sampling method may incorrectly exclude some comparisons, it does not affect the completeness as Algorithm 2 is wrapped in an iterator.

Exclude redundant comparisons. Each comparison c that passes to the second stage of `CandidateComps` will be evaluated on all examples. At this time, if there exists a comparison with a smaller range satisfying the same set of examples with c, c will be excluded from C as its effect is dominated.

Theorem 5.2 shows the completeness of *MetHyl* while solving the synthesis task for Step 1.

Theorem 5.2. *Given program (h, o), a set of instances I and a grammar G for available comparisons, if there exists a keyword preorder R satisfying (1) $(\leq, o) \in R$, (2) $\forall i \in I, CE(R, i) = \emptyset$, and (3) R is constructed by $(\leq, o)$ and comparisons in G, then MetHyl must terminate and return such a keyword preorder.*

5.2 Synthesis Incrementalization

To solve the synthesis task for Step 2 and Step 4, we first rewrite the condition for a query q into Formula 8.

$$(q \circ id)\ e = (?q[q] \circ \mathsf{F}[q](null \triangle ?f_p))\ e \qquad (8)$$

where id is the identity function, i.e., $\lambda x.x$, and $null$ is a dummy function that outputs nothing.

Compared with Definition 3.7, Formula 8 is the specification of partial lifting problem $\mathsf{PLP}(\{id\}, q, null, \cup RE(q, i))$. Therefore, for each query q, *MetHyl* invokes *AutoLifter* to find a solution $(q[q], f_p[q])$ to Formula 8.

Then, *MetHyl* merges these results by fixing $?f_p$ to $f_q \triangle ?f'_p$, where $f_q = \triangle_{q \in Q} f_p[q]$ and Q represents the set of queries. For each query q, such a $?f_p$ and $q[q]$ can be converted into a solution to Formula 8 by adjusting the input type of $q[q]$.

Next, we substitute $?f_p$ into the condition for constructors and rewrite the condition into Formula 9.

$$\forall m \in M, ((f_q \triangle ?f'_p) \circ m)\ e = (?c[m] \circ \mathsf{F}[m](f_q \triangle ?f_p))\ e \quad (9)$$

where M represents the set of constructors.

Compared with Definition 3.7 again, Formula 9 is the specification of lifting problem $\mathsf{LP}(M, f_p, f_p, E)$, where $E[m]$ is equal to $\cup RE(m, i)$. Therefore, the remaining part $?f'_p$ in $?f_p$ and $?c$ can also be synthesized by invoking *AutoLifter*.

5.3 Properties of *MetHyl*

First, Theorem 5.3 guarantees the result of *MetHyl* to be correct on all given instances.

Theorem 5.3. *Given input program (h, o) where h is a relational hylomorphism and a set of instances I, let p^* be the program synthesized by MetHyl. Then $\forall i \in I$, $(h, o) \sim_i p^*$.*

Second, Theorem 5.4 shows that under some assumptions, the program synthesized by *MetHyl* is guaranteed to run in pseudo-polynomial time, which means that the time complexity is polynomial to the size and values in the input.

Theorem 5.4. *Given input program $([\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and a grammar G specifying the program space for synthesis tasks, the program synthesized by MetHyl must be pseudo-polynomial time if the following assumptions are satisfied: (1) ϕ, ψ and programs in G runs in pseudo-polynomial time, (2) each value and the size of each recursive data structure generated by the input program are pseudo-polynomial, (3) all operators in G are linear, i.e., their outputs are bounded by a linear expression with respect to the input.*

6 Implementation

Generating instances. In *MetHyl*, a set of instances is required to (1) extract examples for each synthesis task, and (2) evaluate the efficiency of synthesized preorders.

MetHyl generates instances by sampling. It describes an instance space by assigning each integer with a range and each recursive data structure with an upperbound on the size. Given such a space, *MetHyl* samples instances according to a uniform distribution.

To limit the time cost of executing the input program, *MetHyl* decides this space by iteratively squeezing from a default space until the time cost of the input program on a random instance is smaller than a threshold.

Verifying the result. Due to the complexity of the synthesis result, which involves self-recursion, data structures, and memoization, it is difficult to verify the complete correctness. In our implementation, we use an iterative verifier proposed by Ji et al. [2021b] and guarantee that the probability for the generation error of the synthesis result to be more than 10^{-3} is at most 1.82×10^{-4}.

Grammar. *MetHyl* uses a grammar to describe the space of possible synthesis results. In our implementation, we extended the CLIA grammar in SyGuS-Comp [Padhi et al. 2021] with the following operators related to lists and binary trees.

- Accumulate operator *fold* and lambda expressions.
- Access operators *access* for lists, and *value*, *lchild*, *rchild*, *isleaf* for binary trees.
- Match operator *match* for lists and binary trees, which returns the first occurrence of a sublist (subtree) on a given list (tree). This operator is useful in Step 4, which can correspond the search state to some global data structure.

Others. The original implementation of *AutoLifter* uses *PolyGen* [Ji et al. 2021a], a specialized solver for conditional linear expressions, is used to synthesize $?c$ for lifting problems. Therefore, in our implementation, when the input program uses non-linear operators, we will replace *PolyGen* with a SOTA enumerative solver, *observational equivalence* [Udupa et al. 2013], to make *AutoLifter* applicable.

To implement the preorder synthesizer introduced in Section 5.1, we set n_c^* to 2 and s_c^* to 10^5, which are enough for most known tasks, and set n_t to 8, which ensures that the error rate of `CandidateComps` is at most $e^{-1} \approx 37\%$.

7 Evaluation

Our evaluation answers two research questions.

- **RQ1**: How is the overall performance of *MetHyl*?
- **RQ2**: How do the transformation steps in *MetHyl* perform?

7.1 Dataset

Our evaluation is conducted on a dataset including 37 input programs for 17 different COPs. Besides problem *0/1 knapsack* discussed in Section 2, the other COPs are collected from *Introduction to Algorithms* [Cormen et al. 2009], a widely-used textbook for algorithm courses. This book introduces dynamic programming in its 15th chapter with 4 example COPs and provides 12 other COPs as the exercise. We include all these 16 COPs in our dataset.

To construct input programs, we divide the COPs into two categories according to whether solutions are constrained.

The first category includes COPs where solutions are not constrained. A representative COP here is *rod cutting*, which is the first example in *Introduction to Algorithms*.

Known that a rod of length i worth w_i, the task is to cut a rod of length n into several pieces and maximize the total value.

In this task, all possible ways to cut the rod are valid solutions. For each COP in this category (7 in total), we implement an input program as natural, where the generator g generates all solutions and the scorer o calculates the objective value.

The second category includes COPs where solutions are constrained. A representative COP is *0/1 knapsack*, where a solution is valid only when the total weight is no more than the capacity. For each COP in this category (10 in total), we implement three input programs ($g = [\![\phi, \psi]\!]_{\mathsf{F}}, o$) according to three extreme principles, which correspond to the three programs in Figures 5, 2, 3, respectively.

- The first program keeps all information in the input of ψ and filters out invalid solutions while deciding transitions.
- The second program tries all possibilities of constructing solutions in ψ and lets ϕ filter out invalid ones.
- The last program uses g to generate all solutions and excludes invalid ones via a small enough objective value.

7.2 Experiment Setup

We run *MetHyl* on all 37 tasks in the dataset and set a time limit of 120 seconds for each step in *MetHyl*. If a step in *MetHyl* times out, *MetHyl* skips this step and goes on to the next. For each execution, we record the time cost and the transformation result for each step in *MetHyl*.

7.3 RQ1: Overall Performance of *MetHyl*

We manually verify the results of *MetHyl* and confirm that all synthesized programs are completely correct. The performance of *MetHyl* is summarized in Table 1. For each problem, *COP* lists its name as in *Introduction to Algorithm*, *Imp* lists the index of the principle if the problem is in the second category, T_{inp} lists the time complexity of the input program in $\tilde{\Omega}$ notation, T_{res} lists the time complexity of the result program in O notation, and *Time* lists the total time used by *MetHyl*.

Table 1 demonstrates both the effectiveness and efficiency of *MetHyl*. On effectiveness, *MetHyl* achieves exponential speed-ups against the input program on 36 out of 37 tasks, accounting for 97.3%. Besides, we compare the synthesis results with the reference algorithms provided by *Introduction to Algorithm* and a solution to its exercises [Li 2011], and find that *MetHyl* achieves the same complexity on 26 out of 37 tasks, accounting for 70.3%. On efficiency, the average time cost of *MetHyl* to process a program is only 59.2 seconds.

MetHyl fails in achieving the same complexity as the reference algorithms on 11 tasks because of three reasons.

For task (15-5, 3), *MetHyl* fails because the scorer in this task provides little information. In COP 15-5, a solution is a sequence of editions, and a solution is valid if the results of these editions are equal to the target. In this sense, almost all partial solutions are invalid, on which the scorer in (15-5, 3) simply returns $-\infty$. At this time, *MetHyl* can hardly extract examples for $?R$ and thus fails in the first step. Therefore, the number of solutions remains exponential in the result.

For tasks (15-8, 2) and (15-8, 3), *MetHyl* fails because of useless transitions. In both tasks, ψ generates $O(n)$ transitions but only $O(1)$ among them can lead to valid solutions. However, as *MetHyl* does not change the recursive structure, this non-optimal performance remains in the result and thus leads to extra time costs. Optimizing the recursive structure of hylomorphism is future work.

For the other 8 tasks, *MetHyl* fails because of the enumerative solver. Because the scalability of *observational equivalence* is limited, *MetHyl* times out in Step 2 and 4 when the target program is a complex non-linear program, e.g., $x_1 - \mathrm{dis}(p_1, p_2) + \mathrm{dis}(p_1, p_3) + \mathrm{dis}(p_2, p_3)$ in Task 15-3, where $\mathrm{dis}(p, q)$ is the abbreviation of $(p.x - q.x)^2 + (p.y - q.y)^2$. This fact shows that the effectiveness of *MetHyl* can be further improved by designing efficient solvers for synthesizing $?c$.

Table 1. The overall performance of *MetHyl* on all tasks in the dataset.

COP	Imp	T_{inp}	T_{res}	Time(s)	COP	Imp	T_{inp}	T_{res}	Time(s)	COP	Imp	T_{inp}	T_{res}	Time(s)
0/1 Knapsack	1	2^n	n^2	25.9	15-3		2^n	n^3†	133.4	15-8	1	2^n	n^2	31.3
	2			11.9	15-4	1	2^n	n^3†	141.4		2	n^n	n^3†	25.5
	3			23.8		2			240.5		3			12.1
Rod Cutting		2^n	n^2	18.4		3			240.5	15-9		4^n	n^3	4.8
Matrix Chain		4^n	n^3	15.4	15-5	1	7.6^n	n^2	40.1	15-10		n^n	n^4†	34.1
LCS	1	5.8^n	n^2	6.9		2			41.3	15-11	1	n^n	n^3	142.0
	2			5.9		3	$\approx 7.6^n$†‡		274.9		2		n^4†	137.5
	3			24.3	15-6	1	2^n	n	41.2		3			137.1
Optimal BST		4^n	n^3	24.5		2			28.1	15-12	1	n^n	n^3	54.9
15-1		2^n	n^4†	123.7		3			31.4		2			27.6
15-2	1	2^n	n^2	8.7	15-7	1	n^n	n^3	3.6		3			31.6
	2			5.6		2			10.5					
	3			20.9		3			8.9					

† The result does not achieve the same time complexity as the reference algorithm for the corresponding COP.
‡ The time complexities of both programs are $\tilde{\Theta}(a^n)$, where $\sqrt{a} \approx 2.77$ is the largest real root of $x^4 - 2x^3 - 2x^2 - 1$.

Table 2. The performance of each transformation step.

ID	Exp	Poly	$\perp$	Time	ID	Exp	Poly	$\perp$	Time
1	31	0	1	4.2	2	0	3	7	32.8
3	5	0	0	5.1	4	0	24	3	59.2

7.4 RQ2: Performance of Transformation Steps

We manually analyze all intermediate results of *MetHyl* and confirm that on our dataset, all four steps in the transformation system never increase the time complexity. The performance of each step is summarized in Table 2. For each step, *ID* lists the index, *Exp* lists the number of tasks on which the step achieves an exponential speed-up, *Poly* lists the number of tasks on which the step achieves a polynomial speed-up, $\perp$ lists the number of failed tasks, and *Time* records the average time cost (seconds) on all tasks.

According to Table 2, Step 1 and 3 produce all exponential speed-ups, while Step 2 and 4 produce all polynomial ones. This result matches the target of each step, where the number of solutions (Step 1) and states (Step 3) can be exponential and their scales (Step 2 and 4) are usually polynomial.

Note that the effects of Step 2 and 3 seem to be insignificant in Table 2 because they are usually *delayed* by Step 4. In many cases, the time cost of operating states forms a bottleneck of the time complexity and thus the effects of Step 2 and 3 will not be revealed until Step 4 is finished. For instance, in the example discussed in Section 2, Steps 2, 3, and 4 each reduces an $O(n)$ component to $O(1)$, but the overall time complexity would not change if any of the components remains.

8 Related Work

Program Calculation. *MetHyl* is related to those studies for deriving dynamic programming in program calculation. First, Bird and de Moor [1997]; de Moor [1995]; Morihata et al. [2014]; Mu [2008] derives dynamic programming via *thinning*. All of their approaches are manual and work only for a subclass of hylomorphism, namely *sequential decision procedure*, where the search state is a recursive data structure and the anamorphism simply unfolds the structure of the state. *MetHyl* generalizes *thinning* to general hylomorphism and automates its application via program synthesis.

Second, Liu and Stoller [2003] uses *incrementalization* and *memoization* to derive dynamic programming from simple recursions, which is also a subclass of hylomorphism. Their approach is semi-automated and cannot generalize to general hylomorphism as it does not reduce the scale of the output, which can be exponential. *MetHyl* applies to *incrementalization* to program fragments in the hylomorphism and automates the application via program synthesis.

Last, there are other approaches proposed for deriving dynamic programming [Giegerich et al. 2004; Lin et al. 2021; Pettorossi and Proietti 1996; Pu et al. 2011; Sauthoff et al. 2011]. Most of them are manual or semi-automated. The only automated approach we know is a transformation rule that automatically generates dynamic programming for a sequential decision procedure on lists when several conditions are satisfied [Lin et al. 2021]. However, the scope of this rule is narrow in contrast to hylomorphism: 27 out of 37 programs, accounting for 73.0%, in our dataset are not even sequential decision procedures.

Program Synthesis. There have been many synthesizers proposed for automatically synthesizing algorithms or efficient programs [Acar et al. 2005; Farzan and Nicolet 2021; Fedyukovich et al. 2017; Hu et al. 2021; Knoth et al. 2019; Morita et al. 2007; Smith and Albarghouthi 2016], but none of them are for dynamic programming.

MetHyl uses program synthesizers to (1) synthesize a keyword preorder from negative examples, and (2) synthesize program fragments for incrementalization.

As discussed in Section 5.2, *MetHyl* solves the second task via our previous approach *AutoLifter* [Ji et al. 2021b]. Besides, *AutoLifter* uses a method namely *observational covering* to find a set of key functions covering a set of negative examples, which has a similar form to our first task. However, this method is applicable only when the set of examples is fixed and thus cannot be applied to our first task, where new negative examples will occur while adding key functions.

MetHyl is related to *Relish* [Wang et al. 2018b], a general solver for relational specifications. Both our tasks are instances of relational specifications and thus are in the scope of *Relish*. However, because the *Finite Tree Automata* used by *Relish* does not directly support lambda expressions, *Relish* cannot be applied to our grammar.

MetHyl is also related to the approach proposed by Zhang and Liu [1998], which is a deductive synthesizer for *incrementalization*. However, this approach is available only when the input of queries includes only scalar values and thus cannot be applied to our second task, where the solutions (states) involve recursive data structure.

There are also many mainstream synthesizers for input-output specifications [Balog et al. 2017; Feser et al. 2015; Gulwani 2011; Ji et al. 2020; Osera and Zdancewic 2015; Wang et al. 2018a]. However, such specifications do not exist for either key functions in our first task or fragments in our second task, and thus all these synthesizers are unavailable.

9 Conclusion

In this paper, we propose a novel synthesizer *MetHyl* for automatically synthesizing efficient memoization for relational hylomorphism. Our evaluation demonstrates both the efficiency and the effectiveness of *MetHyl*.

This paper is motivated by the theories of program calculation for deriving efficient dynamic programming algorithms. We notice that there are also studies for deriving other algorithms such as *greedy algorithm* [Bird and de Moor 1993; Helman 1989] and *branch-and-bound* [Fokkinga 1991] from relational hylomorphisms. Extending *MetHyl* to support these algorithms is future work.

References

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

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

Richard Bird and Oege De Moor. 1994. Relational program derivation and context-free language recognition. In *A classical mind: essays in honour of CAR Hoare.* 17–35.

Richard S. Bird and Oege de Moor. 1993. From Dynamic Programming to Greedy Algorithms. In *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report (Lecture Notes in Computer Science, Vol. 755)*, Bernhard Möller, Helmuth Partsch, and Stephen A. Schuman (Eds.). Springer, 43–61. https://doi.org/10.1007/3-540-57499-9_16

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

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* MIT press.

Oege de Moor. 1994. Categories, Relations and Dynamic Programming. *Math. Struct. Comput. Sci.* 4, 1 (1994), 33–69. https://doi.org/10.1017/S0960129500000360

Oege de Moor. 1995. A Generic Program for Sequential Decision Processes. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 982)*, Manuel V. Hermenegildo and S. Doaitse Swierstra (Eds.). Springer, 1–23. https://doi.org/10.1007/BFb0026809

Azadeh Farzan and Victor Nicolet. 2021. Phased synthesis of divide and conquer programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, 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

Maarten M. Fokkinga. 1991. An Exercise in Transformational Programming: Backtracking and Branch-and-Bound. *Sci. Comput. Program.* 16, 1 (1991), 19–48. https://doi.org/10.1016/0167-6423(91)90022-P

Robert Giegerich, Carsten Meyer, and Peter Steffen. 2004. A discipline of dynamic programming over sequence data. *Sci. Comput. Program.* 51, 3 (2004), 215–263. https://doi.org/10.1016/j.scico.2003.12.005

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

Paul Helman. 1989. *A theory of greedy structures based on k-ary dominance relations.* Department of Computer Science, College of Engineering, University of New Mexico.

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2021. Synthesis with Asymptotic Resource Bounds. 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, 783–807. https://doi.org/10.1007/978-3-030-81685-8_37

Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1996. Deriving Structural Hylomorphisms From Recursive Definitions. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 73–82. https://doi.org/10.1145/232627.232637

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

Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 224:1–224:29. https://doi.org/10.1145/3428292

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

Ruyi Ji, Yingfei Xiong, and Zhenjiang Hu. 2021b. Black-Box Algorithm Synthesis âĂŤ Divide-and-Conquer and More. https://jiry17.github.io/pistool/papers/AutoLifter.pdf

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

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 436–449. https://doi.org/10.1145/3192366.3192410

Jian Li. 2011. The solutions to the book âĂIJIntroduction to Algorithm, 3rd EditionâĂİ. http://guanzhou.pub/files/CLRS/CLRS-Part%20Answer.pdf

Shu Lin, Na Meng, and Wenxin Li. 2021. Generating efficient solvers from constraint models. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 956–967. https://doi.org/10.1145/3468264.3468566

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

George B Mathews. 1896. On the partition of numbers. *Proceedings of the London Mathematical Society* 1, 1 (1896), 486–490.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 124–144. https://doi.org/10.1007/3540543961_7

Akimasa Morihata, Masato Koishi, and Atsushi Ohori. 2014. Dynamic Programming via Thinning and Incrementalization. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 186–202. https://doi.org/10.1007/978-3-319-07151-0_12

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

Shin-Cheng Mu. 2008. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, Robert Glück and Oege de Moor (Eds.). ACM, 31–39. https://doi.org/10.1145/1328408.1328414

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

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

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

Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 83–98. https://doi.org/10.1145/2048066.2048076

Georg Sauthoff, Stefan Janssen, and Robert Giegerich. 2011. Bellman's GAP: a declarative language for dynamic programming. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, Peter Schneider-Kamp and Michael Hanus (Eds.). ACM, 29–40. https://doi.org/10.1145/2003476.2003484

Alexander Schrijver. 2003. *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. Springer Science & Business Media.

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

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

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018a. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018b. Relational program synthesis. *PACMPL* 2, OOPSLA (2018), 155:1–155:27.

Yuchen Zhang and Yanhong A. Liu. 1998. Automating Derivation of Incremental Programs. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 350. https://doi.org/10.1145/289423.289480

A Appendix

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

A.1 Proofs for Section 3.4

Theorem A.1 (Theorem 3.6). *Given preorder $R = \{(op_i, k_i)\}$, define function $N_R(S)$ as the following, where $range(k, S)$ is the range of k on S, i.e., $\max_{a\in S}(k\ a) - \min_{a \in S}(k\ a)$, and $\max(S)$ returns the largest element in S with default value 1.*

$$N_R(S) := \left(\prod_i range(k_i, S)\right) \Big/ \max_i \left(range(k_i, S) \;\Big|\; op_i \in \{\leq\}\right)$$

- *For any set S, $|thin\ R\ S| \leq N_R(S)$.*
- *There is an implementation of $(thin\ R)$ with time complexity $O(N_R(S) size(R) + T_R(S))$, where S is the input set, $size(R)$ is the number of comparisons in R, $T_R(s)$ is the time complexity of evaluating all key functions in R for all elements in S.*

Proof. Let $K_\leq$ be the set of key functions in R with operator $\leq$, and let k^* be the key function in $K_\leq$ with the largest range on S, i.e., $\arg\max range(k, S), k \in K_\leq$. Especially, when $K_\leq$ is empty, k^* is defined as the constant function $\lambda x.0$.

Let $K = \{k_1, \dots, k_m\}$ be the set of key functions in R excluding k^*. According to the definition of $N_R(S)$, we have the following equality.

$$N_R(S) = \prod_{i=1}^{m} range(k_i, S)$$

We start with the first claim. Define feature function f_k as $k_1 \triangle \dots \triangle k_m$. Then $N_R(S)$ is the size of f_k's range. By the definition of the keyword preorder, we have the following formula:

$$f_k\ a = f_k\ b \to (aRb \leftrightarrow k^*\ a \leq k^*\ b)$$

In other words, for elements where the outputs of the feature function are the same, their order in R is a total order. Therefore, the number of maximal values in S is no more than the size of the feature function's range, i.e., $N_R(S)$.

Then, for the second claim, Algorithm 3 shows an implementation of *thin*. The time complexity of the first loop (Lines 6-10) is $O(T_R(S))$ and the time complexity of the second loop (Lines 11-20) is $O(N_R(S) size(R))$. Therefore, the overall time complexity of Algorithm 3 is $O(N_R(S) size(R) + T_R(S))$.

The remaining task is to prove the correctness of Algorithm 3. Let $\mathcal{A}_x$ be the algorithm weakened from Algorithm 3 by replacing the loop upper bound in Line 11 from m to x. Besides, let R_x be the keyword preorder $\{(op_i, id)_{i=1}^x\} \cup \{(=, id)_{i=x+1}^m\}$. Now, consider the following claim.

- After running $\mathcal{A}_x$ on set S, the value of $Val[w]$ is equal to $\arg\max_a k^*\ a, a \in S \land wR_x(f_k\ a)$. If there is no such a exist, $Val[w]$ is equal to $\perp$.

If this claim holds, after running $\mathcal{A}_m$, i.e., Algorithm 3 on S, $Val[f_k\ a] = a$ if and only if a is a local maximal in S. Therefore, we get the correctness of Algorithm 3.

Algorithm 3: An implementation of *thin R*.

```
Input: A set $S$ of elements.
Output: A subset including all maximal values in $S$.
1  Extract $k^*$ and $f_k = k_1 \triangle \dots \triangle k_m$ from $R$;
2  $op_i \leftarrow$ the operator corresponding to $k_i$;
3  $[mi_i, ma_i] \leftarrow$ the range of $k_i$ on $S$;
4  $\mathbb{W} \leftarrow [mi_1, ma_1] \times [mi_2, ma_2] \times \dots \times [mi_m, ma_m]$;
5  $\forall w \in \mathbb{W}, Val[w] \leftarrow \perp$;
6  foreach $a \in S$ do
7      if $Val[f_k\ a] = \perp \lor k^*\ (Val[f_k\ a]) \leq k^*\ a$ then
8          $Vak[f_k\ a] \leftarrow a$;
9      end
10 end
11 foreach $i \in [1, m]$ do
12     if $op_i \in \{=\}$ then continue;
13     foreach $w \in \mathbb{W}$ in the decreasing order of $w.i$ do
14         if $w.i = mi_i \lor Val[w] = \perp$ then continue;
15         $w' \leftarrow w$;    $w'.i \leftarrow w.i - 1$;
16         if $Val[w'] = \perp \lor k^*\ Val[w'] \leq k^*\ Val[w]$ then
17             $Val[w'] \leftarrow Val[w]$;
18         end
19     end
20 end
21 return $\{a \mid a \in S \land Val[f_k\ a] = a\}$;
```

To prove this claim, we make an induction on the value of m. First, when m is equal to 0, this claim holds because only the element with the largest output of k^* is retained while initializing *Val* (Lines 6-10).

Then, for any $x \in [1, m]$, assume that the claim holds for $\mathcal{A}_{x-1}$. When op_x is equal to $=$, the correctness of $\mathcal{A}_{x-1}$ directly implies the correctness of $\mathcal{A}_x$. Therefore, we consider only the case where $op_x \in \{\leq\}$ below.

Let Val' be the value of *Val* after running $\mathcal{A}_{x-1}$, and let *Val* be the value of *Val* after running $\mathcal{A}_x$. For any $w \in W$ and $i \in [mi_x, ma_x]$, let w_i be the feature that $\forall j \neq x, w_i.j = w.j$ and $w_i.x = i$. According to Lines 11-20 in Algorithm 3, $Val[w]$ is equal to the element with the largest output of k^* among $Val'[w_{w.i}], Val'[w_{w.i+1}], \dots, Val'[w_{ma_x}]$. (For simplicity, we define $k^*\ \perp$ as $-\infty$).

Assume that the claim does not hold for $\mathcal{A}_x$. Then, there exists $w \in \mathbb{W}$ and $a \in S$ satisfying the following formula.

$$\begin{aligned} &k^*\ Val[w] < k^*\ a \land wR_x(f_k\ a) \\ \Longrightarrow\ &k^*\ Val'\left[w_{k_x\ a}\right] < k^*\ a \land w_{k_x\ a}R_{x-1}(f_k\ a) \end{aligned}$$

This fact contradicts with the inductive hypothesis and thus the induction holds, i.e., the claim holds for all $\mathcal{A}_x$. ◻

A.2 Proofs for Section 4.1

In Section 4.1, for simplicity, we assume that each transition, i.e., each element in the output of ϕ, involves at most one search state. Therefore, in this section, we first extend Section

4.2 to general relational hylomorphisms and then prove the generalized lemmas.

Let $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ be the input program. We first generalize the concept of $\twoheadrightarrow_s$. When there is a transition involving multiple states, a partial solution may be constructed from multiple partial solutions. For any state s, let $s_1, \dots, s_n$ be a series of states in $T_h\ s$, and $p_1, \dots, p_n$ be a series of partial solutions where $\forall i \in [1, n], p_i \in h\ s_i$. We use $(p_1, \dots, p_n) \twoheadrightarrow_s p$ to denote that p can be constructed from partial solutions $p_1, \dots, p_n$, i.e., there exists an invocation of ϕ in $h\ s$ where the input includes $p_1, \dots, p_n$ and the output is p.

Using the generalized notation $\twoheadrightarrow$, we generalize Lemma 4.1 to general relational hylomorphisms as the following.

Lemma A.2 (Lemma 4.1). *Given instance i and program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$, $(rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o) \sim_i ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$ for keyword preorder $?R$ if $?R$ satisfies that (1) $(\leq, o) \in ?R$, and (2) for any state $s \in S_h\ i$, any sequence of states $s_1, \dots, s_n \in (T_h\ s)$, and any two sequences of partial solutions $\overline{p_1} = (p_{1,1}, \dots, p_{1,n})$ and $\overline{p_2} = (p_{2,1}, \dots, p_{2,n})$ where $p_{1,i}, p_{2,i} \in h\ s_i$, $\wedge_{i=1}^n p_{1,i} ?R p_{2,i}$ implies that partial solutions generated from $\overline{p_1}$ are dominated by partial solutions generated from $\overline{p_2}$ in the sense of $?R$, i.e.,*

$$\bigwedge_{i=1}^{n} p_{1,i} ?R p_{2,i} \rightarrow \forall p_1', \left(\overline{p_1} \twoheadrightarrow_s p_1' \rightarrow \exists p_2', \overline{p_2} \twoheadrightarrow_s p_2' \wedge p_1' ?R p_2'\right) \tag{10}$$

Proof. For simplicity, we use r to denote $rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}$. Because ψ in h is also used in r, the search trees generated by r and h on instance i are exactly the same.

For simplicity, we use $S_1 \sqsupseteq_R S_2$ to denote that elements in S_1 dominates elements in S_2 in the sense of preorder R, i.e., $\forall a \in S_2, \exists b \in S_1, aRb$. By the definition of *thin*, for any preorder R and any set S, $thin\ R\ S \sqsupseteq_R S$ always holds.

Let us consider the following claim.

- For any state s in $S_h\ i$, $r\ s \subseteq h\ s \wedge r\ s \sqsupseteq_{?R} h\ s$.

Let p_o be any solution with the largest objective value in $h\ i$. If this claim holds, there must be a solution p^* in $r\ i$ such that $p_o ?R p^*$. By the precondition that $(\leq, o) \in ?R$, $o\ p^* \geq o\ p_o$. Because $p^* \in r\ i \subseteq h\ i$, we have $o\ p^* = o\ p_o$. Therefore, at least one solution with the largest objective value are retained in $r\ i$, which implies that $(r, o) \sim_i (h, o)$.

We prove this claim by structural induction on the search tree. First, $r\ s \subseteq h\ s$ can be obtained by the definition of rg and $[\![\phi, \psi]\!]_{\mathsf{F}}$. Let us unfold the definition of h and r.

$$\begin{aligned} h &= cup \circ \mathsf{P}\phi \circ cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}h) \circ \psi \\ r &= thin ?R \circ cup \circ \mathsf{P}\phi \circ cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}r) \circ \psi \end{aligned}$$

Starting from the inductive hypothesis, we have the following derivation.

$$\begin{aligned} &\forall s' \in T_h\ s, r\ s' \subseteq h\ s' \\ \Longrightarrow\ &\forall t \in \psi\ s, (car[\mathsf{F}] \circ \mathsf{F}r)\ t \subseteq (car[\mathsf{F}] \circ \mathsf{F}h)\ t \\ \Longrightarrow\ &(cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}r) \circ \psi)\ s \subseteq \\ &\qquad (cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}h) \circ \psi)\ s \\ \Longrightarrow\ &r\ s \subseteq h\ s \end{aligned}$$

By the induction, we prove that $\forall s \in S_h\ i, r\ s \subseteq h\ s$.

The remaining task is to prove $\forall s \in S_h\ i, r\ s \sqsupseteq_{?R} h\ s$. For any state s, let P_s be the set of partial solutions constructed in $r\ s$ before applying *thin* $?R$. Let us consider another claim.

- For any state s in $S_h\ i$, $P_s \sqsupseteq_{?R} h\ s$.

If the second claim holds, we prove the first claim by

$$r\ s = thin\ ?R\ P_s \sqsupseteq_{?R} P_s \sqsupseteq_{?R} h\ s$$

Therefore, we need only to prove the second claim via the inductive hypothesis. Suppose this claim does not hold for state s.

$$P_s \not\sqsupseteq_{?R} h\ s \implies \exists p \in h\ s, \forall p' \in P_s, \neg p ?R p' \tag{11}$$

Suppose partial solution p is constructed from partial solutions $p_1, \dots, p_n$ where p_i is taken from state s_i. By the inductive hypothesis, for each $i \in [1, n]$, there exists $p_i' \in r\ s_i$ such that $p_i ?R p_i'$. Let $\overline{p} = (p_1, \dots, p_n)$ and $\overline{p'} = (p_1', \dots, p_n')$.

$$\begin{aligned} \bigwedge_{i=1}^{n} p_i ?R p_i' &\implies \forall p_1', \left(\overline{p} \twoheadrightarrow_s p_1' \rightarrow \exists p_2', \overline{p'} \twoheadrightarrow_s p_2' \wedge p_1' ?R p_2'\right) \\ &\implies \exists p_2', \overline{p'} \twoheadrightarrow_s p_2' \wedge p ?R p_2' \\ &\implies \exists p_2' \in P_s, p ?R p_2' \end{aligned} \tag{12}$$

Formula 12 contradicts with Formula 11. Therefore, we prove the second claim, and thus the induction holds. □

After the generalization, given a preorder R and an instance i, a set of counterexamples $CE(R, i)$ for R can also be extracted from Formula 10. A counter example is a sequence of pairs of partial solutions $((p_{1,j}, p_{2,j})_{j=1}^n)$, representing that $\neg p_{1,j} R p_{2,j}$ is expected to hold for at least one $j \in [1, n]$.

The following is the generalized version of Lemma 4.2.

Lemma A.3 (Lemma 4.2). *Given instance i, for any keyword preorders $R_1 \subseteq R_2$, the following formula is always satisfied.*

$$\forall e \in CE(R_1, i), e \notin CE(R_2, i) \leftrightarrow \exists (p_1, p_2) \in e, \neg p_1 (R_2/R_1) p_2$$

where R_2/R_1 represents the keyword preorder formed by the new comparisons in R_2 compared with R_1.

Proof. We start with the $\leftarrow$ direction. Suppose there is an example e in $CE(R_1, i)$ satisfying $\exists (p_1, p_2) \in e, \neg p_1 (R_2/R_1) p_2$. Because the comparisons in R_2/R_1 is a subset of R_2, this precondition implies that $\exists (p_1, p_2) \in e, \neg p_1 R_2 p_2$. By Formula 10, e cannot be a counter example for R_2, i.e., $e \notin CE(R_2, i)$.

For the $\rightarrow$ direction, suppose there is an example e in $CE(R_1, i)$ such that $\forall (p_1, p_2) \in e, p_1 (R_2/R_1) p_2$.

Let $(p_{1,1}, p_{2,1}), \dots, (p_{1,n}, p_{2,n})$ be all pairs in example e, let $\overline{p_1}$ and $\overline{p_2}$ be the sequences of $p_{1,j}$ and $p_{2,j}$ respectively. By the definition of CE, we have (1) $\forall (p_1, p_2) \in e, p_1 R_1 p_2$, and (2) the following formula.

$$\begin{aligned} &\exists p_1', \overline{p_1} \twoheadrightarrow_s p_1' \wedge \forall p_2', \left(\overline{p_2} \twoheadrightarrow_s p_2' \rightarrow \neg p_1' R_1 p_2'\right) \\ \Longrightarrow\ &\exists p_1', \overline{p_1} \twoheadrightarrow_s p_1' \wedge \forall p_2', \left(\overline{p_2} \twoheadrightarrow_s p_2' \rightarrow \neg p_1' R_2 p_2'\right) \qquad (13) \end{aligned}$$

By the definition of keyword preorders, we have the following derivation.

$$\begin{aligned} &\forall (p_1, p_2) \in e, p_1 R_1 p_2 \wedge \forall (p_1, p_2) \in e, p_1 (R_2/R_1) p_2 \\ \Longrightarrow\ &\forall (p_1, p_2) \in e, \forall (op, k) \in R_2, (k\ p_1) op (k\ p_2) \\ \Longrightarrow\ &\forall (p_1, p_2) \in e, p_1 R_2 p_2 \qquad (14) \end{aligned}$$

Combining Formula 14 with 13, we know example e is in $CE(R_2, i)$, and the other direction of this lemma is proved. □

A.3 Proofs for Section 4.2

Lemma A.4 (Lemma 4.3). *Given instance i and program $prog_2$ in Form 3, let $prog_2'$ be the program constructed in Step 2. $prog_2 \sim_i prog_2'$ if for any query q and any constructor m, Formula 6 and Formula 7 are satisfied respectively.*

$$\forall e \in RE(q, i), q\ e = ?q[q]\ (\mathsf{F}[q] ?f_p\ e) \qquad (15)$$

$$\forall e \in RE(m, i), ?f_p\ (m\ e) = ?c[m]\ (\mathsf{F}[m] ?f_p\ e) \qquad (16)$$

Proof. Recall the form of $prog$ and $prog_2'$ as the following.

$$\begin{aligned} prog_2 &= (rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o) \\ prog_2' &= (rg((thin\ R') \circ cup \circ \mathsf{P}\phi', \psi)_{\mathsf{F}}, ?q[o]) \end{aligned}$$

Comparing $prog_2'$ with $prog_2$, there are several expression-level differences: (1) all key functions in $?R$ are replaced with the corresponding $?q$, (2) the objective function is replaced with $?q[o]$, (3) all solution-related functions in ϕ are replaced with the corresponding $?q$ and $?c$.

Let e_1, e_2 be the small-step executions of $prog_2$ and $prog_2'$ on instance i, and let $e[k]$ be the kth program in execution e. Let us consider the following claim.

- For any k, $e_1[k]$ will be exactly the same with $e_2[k]$ after (1) replacing all solution-related functions with the corresponding $?q$ and $?c$, and (2) replacing all solutions with the corresponding outputs of $?f_p$.

If this claim holds, the last programs in e_1 and e_2 must be exactly the same because they are the outputs of $prog_2$ and $prog_2'$ and include neither functions nor solutions. Therefore, this claim implies that $prog_2 \sim_i prog_2'$.

We prove this claim by induction on the number of steps. When $k = 0$, the claim directly holds because there is no solution constructed and the correspondence of functions is guaranteed by the construction of $prog_2$.

Then for any $k > 0$, consider the kth evaluation rule applied to e_1 and e_2. By the inductive hypothesis, these two evaluation rules must be the same.

- If this evaluation rule relates to partial solutions, it must be the evaluation of a solution-related function. By the inductive hypothesis, (1) the scalar values in both inputs are exactly the same, and (2) the partial solutions used in e_2 are equal to the outputs of $?f_p$ on the partial solutions used in e_1. At this time, the examples used in the synthesis task ensure that the evaluation result is still corresponding.
- If this evaluation rule does not relate to partial solutions, by the inductive hypothesis, the evaluation in e_1 and e_2 must be exactly the same.

Therefore, the induction holds, and thus the claim holds. □

A.4 Proofs for Section 4.3

Lemma A.5 (Lemma 4.4). *Given instance i and program (r, o), where r is a recursive generator, $(r^{?f_m}, o) \sim_i (r, o)$ if for any states $s_1, s_2 \in (S_r\ i)$, $(?f_m\ s_1 = ?f_m\ s_2) \rightarrow (r\ s_1 = r\ s_2)$.*

Proof. Consider the following claim.

- Each time when $r^{?f_m}\ s$ returns, (1) the result is equal to $r\ s$, and (2) for any state $s' \in S_r\ i$, there is result recorded with keyword $?f_m\ s'$ implies that the results is $r\ s'$.

If the claim holds, the lemma is obtained by $r^{?f_m}\ i = r\ i$.

Let $r^{?f_m}\ s_1, \dots, r^{?f_m}\ s_n$ be all invocations of $r^{?f_m}$ during $r^{?f_m}\ i$ and they are ordered according to the exit time. We prove the claim by induction on the prefixes of this sequence. For the empty prefix, the claim holds as the memoization space is empty.

Now, consider the kth invocation $r^{?f_m}\ s_k$. There are two cases. In the first case, there has been a corresponding result recorded in the memoization space. At this time, by the inductive hypothesis, this result must be equal to $r\ s_k$, and thus the claims still hold when $r^{?f_m}\ s_k$ returns.

In the second case, there has not been a corresponding result recorded. By the inductive hypothesis, the results of the recursions made by $r^{?f_m}\ s_k$ must be the results of the corresponding recursions made by $r\ s_k$. Therefore, the execution of $r^{?f_m}\ s_k$ must be exactly the same with $r\ s_k$ and thus $r^{?f_m}\ s_k = r\ s_k$. By the examples used to synthesize $?f_m$, we know that for any other state $s \in S_r\ i$, $?f_m\ s = ?f_m\ s_k$ implies that $r\ s = r\ s_k$, i.e., the newly memoized result.

Therefore, the induction holds, and thus the claim holds. □

A.5 Proofs for Section 5.1

Similar to Section 4.1, we generalize Lemma 5.1 to general relational hylomorphisms as the following.

Lemma A.6 (Lemma 5.1). *Given a set of instances I, for any keyword preorders $R_1 \subseteq R_2$ where $\forall i \in I, CE(R_2, i) = \emptyset$, there exists a comparison $(op, k) \in R_2/R_1$ satisfying at least $1/(|R_2|-$*

$|R_1|$) *portion of examples in* $CE(R_1, I) = \cup_{i\in I} CE(R_1, i)$*, i.e.,*

$$\left|\left\{e \in CE(R_1, I) \;\middle|\; \exists (p_1, p_2) \in e, \neg\Big((k\ p_1) op (k\ p_2)\Big)\right\}\right| \geq \frac{|CE(R_1, I)|}{|R_2| - |R_1|}$$

Proof. Let $(op_1, k_1), \dots, (op_n, k_n)$ be comparisons in R_2/R_1. Define keyword preorders R_x^p as $R_1 \cup \{(op_j, k_j)_{j=1}^x\}$, and define R_x^a as $R_1 \cup \{(op_x, k_x)\}$. By the definition of keyword preorders, this lemma is equivalent to the following formula.

$$\exists x \in [1, n], \left|CE(R_1, I)\middle/CE(R_x^a, I)\right| \geq \frac{|CE(R_1, I)|}{n} \qquad (17)$$

We prove Formula 17 in two steps. First, we prove that all $x \in [1, n]$ satisfies the following formula.

$$\left|\Big(CE(R_1, I)/CE(R_x^p, I)\Big)\middle/\Big(CE(R_1, I)\middle/CE(R_{x-1}^p, I)\Big)\right| \leq \left|CE(R_1, I)\middle/CE(R_x^a, I)\right| \qquad (18)$$

For any x, let C_x^p be the set in the left-hand side and let C_x^a be the set in the right-hand side. By Lemma A.3, the following derivation holds for any $e \in CE(R_1, I)$.

$$\begin{aligned} e \in C_x^p \iff & \forall (p_1, p_2) \in e, \forall j \in [1, x-1], (k_j\ p_1) op_j (k_j\ p_2) \\ & \wedge \exists (p_1, p_2) \in e, \exists j \in [1, x], \neg (k_j\ p_1) op_j (k_j\ p_2) \\ \implies & \exists (p_1, p_2) \in e, \neg (k_x\ p_1) op_x (k_x\ p_2) \\ \iff & e \in C_x^a \end{aligned}$$

Therefore, $|C_x^p| \leq |C_x^a|$ and thus Formula 18 is proved.

Then, we prove the following formula.

$$\exists x \in [1, n], |C_x^p| \geq \frac{|CE(R_1, I)|}{n} \qquad (19)$$

Because $CE(R_2, I) = \emptyset$, we know $C_1^p \cup C_x^p \cup \dots \cup C_n^p = CE(R_1, I)$. Therefore, $\sum_{i=1}^n |C_i^p| \geq |CE(R_1, I)|$. Let x^* be the index where $|C_x^p|$ is maximized.

$$n\left|C_{x^*}^p\right| \geq \sum_{i=1}^n \left|C_i^p\right| \geq |CE(R_1, I)|$$

Therefore, we prove that Formula 19 holds for $x = x^*$.

As the combination of Formula 18 and Formula 19 directly implies Formula 17, the target lemma is proved. □

Theorem A.7 (Theorem 5.2)**.** *Given program (h, o), a set of instances I and a grammar G for available comparisons, if there exists a keyword preorder R satisfying (1) $(\leq, o) \in R$, (2) $\forall i \in I, CE(R, i) = \emptyset$, and (3) R is constructed by $(\leq, o)$ and comparisons in G, then MetHyl must terminate and return such a keyword preorder.*

Proof. Let R be any solution satisfying the three conditions. According to Algorithm 2, given a finite set of comparisons and a size limit, function `BestPreorder` always terminates.

We name an invocation of `BestPreorder` good if the comparison space including all comparisons except $(\leq, o)$ used in R and n_c is no smaller than $size(R) - 1$. According to the iteration method used to decide C and n_c, for any t, there will be t good invocations finished within finite time.

Let $(op_1, k_1), \dots, (op_n, k_n)$ be an order of comparisons in $R/\{(\leq, o)\}$ such that for any $x \in [1, n]$, (op_x, k_x) will be a valid comparison for `CandidateComps` in the xth turn if $(op_1, k_1), \dots, (op_{x-1}, k_{x-1})$ are selected in the previous terms. According to Lemma A.6, such an order must exist.

Suppose the error rate of `CandidateComps` is at most c, i.e., the probability for `CandidateComps` to exclude a valid comparison is at most c. For a good invocation of `BestPreorder`, R will be found if for any $x \in [1, n]$, (op_x, k_x) is not falsely excluded in the xth turn by `CandidateComps`. Therefore, the probability for R to be found in a good invocation is at least $c' = (1-c)^n$, which is a constant.

So, the probability for *MetHyl* not to terminate after t good invocations is at most $(1-c')^t$. When $t \to +\infty$, this probability converges to 0. □

A.6 Proofs for Section 5.3

Theorem A.8 (Theorem 5.3)**.** *Given input program (h, o) where h is a relational hylomorphism and a set of instances I, let p^* be the program synthesized by MetHyl. Then $\forall i \in I$, $(h, o) \sim_i p^*$.*

Proof. Because the correctness of Step 4 can be proved in the same way as Step 2, this theorem is directly from Lemma 4.1, 4.3, 4.4, and the correctness of Step 4. □

Theorem A.9 (Theorem 5.4)**.** *Given input $([\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and a grammar G specifying the program space for synthesis tasks, the program synthesized by MetHyl must be pseudo-polynomial time if the following assumptions are satisfied: (1) ϕ, ψ and programs in G runs in pseudo-polynomial time, (2) each value and the size of each recursive data structure generated by the input program are pseudo-polynomial, (3) all operators in G are linear, i.e., their outputs are bounded by a linear expression with respect to the input.*

Proof. The time complexity of the resulting program can be decomposed into four factors: (1) the number of recursive invocations on the generator, (2) the maximum number of partial solutions returned by each invocation, (3) the time complexity of each invocation of the generator, and (4) the time complexity of each invocation of the scorer. To prove this theorem, we only need to prove that all of these four factors are pseudo-polynomial.

First, we prove that for any program in G that returns a scalar value, its range is always pseudo-polynomial. For any such program p in G, let $f_p(n, w)$ be a polynomial representing that the time cost of p is at most $f_p(n, w)$ when n scalar values in range $[-w, w]$ are provided as the input.

By the third precondition, there exists a constant c such that for any operator $\oplus$ in G, for any input $\overline{x}$ and any output value $y \in \oplus\overline{x}$, $|y|$ is always at most $c \sum_{x \in \overline{x}} |x|$.

Suppose the size of program p is s_p, which is a constant while analyzing the complexity of p. Now, suppose n scalar values in range $[-w, w]$ are provided as the input to p. After executing the first operator, the sum of all available values is at most $f_p(n, w) \times cnw$, because there are at most $f_p(n, w)$ values due to the time limit and each value is at most cnw according to the third precondition. Then, after the second operator, this sum increases to $f_p(n, w) \times c(f_p(n, w) \times cnw) = c^2 f_p(n, w)^2 \times nw$. In this way, we know that after executing all s_p operators, the sum of all available values is at most $c^{s_p} f_p(n, w)^{s_p} \times nw$. Because s_p is a constant, this upper bound is still pseudo-polynomial with respect to the input.

Second, we prove that the first two factors are pseudo-polynomial. The first factor is bounded by the size of $?f_m$'s range, which is bounded by the product of the sizes of the ranges of key functions in $?f_m$. The second factor is bounded by the number of partial solutions returned by $\mathit{thin}\ ?R$. By Theorem 3.6, this value is also bounded by the product of the sizes of the ranges of key functions in $?R$. Because the number of key functions in $?f_m$ and $?R$ are constants, we only need to prove that the size of each key function's range is pseudo-polynomial.

- For key functions in $?f_m$, by the second precondition, in the input program, both the size of a state and values in a state are pseudo-polynomial with respect to the global input. By our first result, we obtain that the size of the range of each key function in $?f_m$ is pseudo-polynomial.
- For key functions in $?R$, by the second precondition, in the input program, both the size of a partial solution and values in a partial solution are pseudo-polynomial. By our first result, the scale of the new partial solution, i.e., the output of $?f_p$, must also be pseudo-polynomial. By the first result again, we obtain that the size of the range of each key function in $?R$ is pseudo-polynomial.

Third, we prove that the third factor is pseudo-polynomial. According to Section 4, the generator in the resulting program must be in the following form:

$$rg((\mathit{thin}\ ?R) \circ \mathit{cup} \circ \mathsf{P}\phi', \psi')$$

Therefore, the time complexity of each invocation can be further decomposed into four factors: (3.1) the time cost of $\mathit{thin}\ ?R$, (3.2) the time cost of ϕ', (3.3) the time cost of ψ', and (3.4) the number of invocations of ϕ'.

- According to Theorem 3.6, Factor 3.1 is bounded by the sizes of the ranges of the key functions in $?R$, which has been proven to be pseudo-polynomial.
- For Factor 3.1 (3.2), the time cost of ϕ' (ψ') is bounded by the time cost of ϕ (ψ) and all inserted program fragments $?q$ and $?c$ in Step 2 (Step 4). By the first precondition, their time costs are all pseudo-polynomial with respect to the new state (the new partial solution), which has also been proven to be pseudo-polynomial in both values and scale. Therefore, the time cost of ϕ' (ψ') is pseudo-polynomial.
- For Factor 3.3, by the first condition, the number of transitions (denoted as n_t) is pseudo-polynomial. The number of partial solutions returned by each recursive invocation (denoted as n_p) has been proven to be pseudo-polynomial, and the number of states (denoted by n_s) involved by a single transition is a constant. Therefore, the number of invocations of ϕ', which is bounded by $n_t \times n_p^{n_s}$, is also pseudo-polynomial.

Therefore, we prove that the third factor is also pseudo-polynomial with respect to the global input.

At last, the fourth operator is pseudo-polynomial because (1) the number of solutions and the scale of solutions are both pseudo-polynomial, and (2) the time complexity of the new objective function, which is a program in G, is pseudo-polynomial by the first precondition.

In summary, all four factors are pseudo-polynomial, and thus we prove the target theorem. □