Automating Thinning Theorem: Synthesizing Efficient Dynamic Programming Algorithms

RUYI JI, Peking University, China
YUWEI ZHAO, Peking University, China
YINGFEI XIONG*, Peking University, China
ZHENJIANG HU, Peking University, China

Dynamic programming is an important optimization technique, but designing efficient dynamic programming algorithms can be difficult for even professional programmers. Thinning, a technique developed for systematically deriving efficient dynamic programming algorithms, has received much attention in studies because of its effectiveness for a large class of problems. Despite the success of thinning in theory, its practical usage is still limited because (1) applying thinning requires mathematical and algorithmic background, and (2) applying thinning solely may not be enough to generate algorithms as efficient as proposed by human experts.

In this paper, we propose two approaches, *MetHyl* and *MetHyl*$^+$, to resolve both problems. First, *MetHyl* automates the application of thinning via program synthesis, and thus eliminates the burden to the user for applying thinning. Second, *MetHyl*$^+$ integrates three rules into *MetHyl* that optimizes three important factors on the time complexity of dynamic programming algorithms that are ignored by thinning, and thus make it able to automatically generate expert-level dynamic programming algorithms on many tasks.

We evaluate our approaches on 37 tasks related to 16 optimization problems collected from *Introduction to Algorithm*, a popular textbook for algorithm courses. The results show that *MetHyl*$^+$ achieves exponential speed-ups on 97.3% tasks with an average time cost of less than one minute. Moreover, *MetHyl*$^+$ generates algorithms that are as efficient as the reference programs provided by human experts on 70.3% tasks.

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 valid 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, where 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 trivial, different memoization algorithms could have huge performance differences. Designing an *efficient* memoization algorithm for a specific problem is difficult and takes algorithmic efforts.

*Corresponding author

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

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

Motivated by the importance and difficulty of designing efficient memoization, many approaches have been proposed for systematically transforming a plain program into an efficient memoization algorithm. In this paper, we consider one important approach among them, namely *thinning* [Bird and de Moor 1997]. Thinning transforms a plain program specified by a recursive generator, which generates all valid solutions, and an objective function, which evaluates the objective value for each solution, into a more efficient program that does not consider most solutions. By previous studies [Bird 2001; Bird and de Moor 1997; de Moor 1995; Morihata 2011; Morihata et al. 2014; Mu 2008; Sasano et al. 2000], thinning can derive efficient memoization for a large class of COPs.

However, despite the success of thinning in theory, its help to average programmers is still limited because of two shortages. First, grasping the usage of thinning requires mathematical and algorithmic background. On the one hand, the formal definition of thinning is highly abstracted and involves concepts in the category theory. On the other hand, thinning requires the user to provide a proper preorder on solutions, and in many cases, finding such a preorder is non-trivial and relies on algorithmic intuitions. Therefore, learning and using thinning are both difficult for most programmers.

Second, though thinning can generate efficient memoization algorithms for many COPs, applying this approach solely is usually not enough to achieve an algorithm that is as efficient as the one proposed by human experts. In our evaluation, the time complexity of the memoization derived by thinning is asymptotically larger than the reference solutions on 34/37(91.9%) tasks in our dataset.

In this paper, we make two contributions to resolve these shortages respectively. For the first shortage, to remove the burden from the user, we show that the application of thinning can be fully automated via program synthesis. *The first contribution of this paper is a fully automated approach for thinning, namely MetHyl. MetHyl* treats the application of thinning as a program synthesis task for the preorder and follows the framework of programming-by-example [Shaw et al. 1975]. Given a recursive generator, an objective function, and several concrete instances of the COP, *MetHyl* extracts a set of examples for the preorder according to the theory of thinning, where each example specifies that the effectiveness of one solution is not dominated by another. *MetHyl* synthesizes a valid preorder from the examples via a novel synthesis algorithm, and then automatically generates an efficient memoization algorithm by thinning with the synthesized preorder. To use *MetHyl*, the user needs neither to find out a proper preorder him/herself nor to learn anything about thinning. In this way, the difficulty of using thinning in practice is greatly reduced.

We implement *MetHyl* and evaluate it on 37 tasks collected from *Introduction to Algorithm* [Cormen et al. 2009], a popular textbook for algorithm courses. The results show that (1) *MetHyl* successfully synthesizes a preorder for thinning on 36/37(97.3%) tasks with an average time cost of 4.21 seconds, and (2) the program generated by *MetHyl* achieves exponential speed-ups against the plain program on 31/37(83.8%) tasks. As mentioned before, we also compare the results of *MetHyl* with the reference solutions to COPs in *Introduction to Algorithm* provided by Cormen et al. [2009] and Li [2011]. The results demonstrate the gap between thinning and human experts: *MetHyl* achieves the same time complexity as the reference solution on only 3/37(8.1%) tasks.

For the second shortage, to further improve the memoization generated by *MetHyl*, we analyze the time complexity of memoizing a recursive generator and show that it is determined by four different factors: (1) the number of solutions returned by the generator, (2) the time cost of constructing solutions from the recursive results, (3) the number of memoized search states, and (4) the time cost of constructing recursive search states from the current one. The main shortage of thinning is that it focuses only on the first factor, while a human expert can make a comprehensive optimization on all four factors. Therefore, we also consider the other three factors. *The second contribution of this paper is three other rules and their automation for the remaining three factors.*

- For the second factor, to reduce the time cost of constructing solutions, our rule replaces the solutions in the plain program, which usually involves inductive data structures, into a tuple comprising a small number of scalar values while keeping the behavior unchanged. In this way, the time cost is greatly reduced, as manipulating a tuple is usually much faster than an inductive data structure.
- For the third factor, to reduce the number of memoized search states, our rule requires a proper equivalence relation over search states and optimizes by skipping those search states of which an equivalent search state has been memoized before.
- The rule for the fourth factor is similar to the rule for the second factor. It reduces the time cost of constructing search states by replacing them with tuples.

Similar to thinning, the automation of these rules also involves program synthesis and follows the framework of programming-by-example. In theory, we show the effectiveness of these three rules: Under certain assumptions, thinning and the three rules together are guaranteed to reduce the time complexity of the input program to pseudo-polynomial.

We integrate these rules into *MetHyl* as *MetHyl*$^+$ and evaluate it on our dataset. The results demonstrate that the improvement brought by the three rules is significant. First, *MetHyl*$^+$ achieves exponential speed-ups on 36/37(97.3%) tasks with an average time cost of 59.2 seconds. Second, on 26/37(70.3%) tasks, *MetHyl*$^+$ achieves the same time complexities as the reference solutions.

2 OVERVIEW

In this section, we introduce the main ideas of thinning, *MetHyl*, and *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, $xs = [(3, 3), (2, 2), (1, 2)], W = 4$ describes an instance of 0/1 knapsack, where three items are available, their weights are 3, 2, 1 respectively, their values are 3, 2, 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.

In this section, we assume that the number of the items (i.e., $|xs|$) and the capacity (i.e., W) are on the same magnitude, denoted by $O(n)$. At this time, there is a standard dynamic programming algorithm for 0/1 knapsack, which runs in $O(n^2)$ time.

2.1 Problem Specification and Memoization

To formally describe a COP, we need to specify (1) the set of valid solutions, and (2) the objective value of each solution. In a natural specification of 0/1 knapsack, the set of valid solutions is all subsets of items whose total weight is within the capacity, and the objective value is the total value of items in the subset.

In this paper, we assume these two parts are specified by two programs g and o respectively.

- The generator g takes the parameters of the problem (in this case, the list of items) as the 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 in Figure 1 shows one such specification (g, o) for 0/1 knapsack. The parameter g of function g is itself to enable recursion later with a fixed-point combinator and the parameter xs 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 of items whose total weight is within the capacity, and o calculates the total value of items. For

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

Fig. 1. One specification (g, o) for 0/1 knapsack, where W is a global variable representing the capacity.

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

Fig. 2. A template for transforming programs (g, o) to a memoization algorithm.

simplicity, we shall directly use g to refer to the recursive version. Note that such a specification may not be unique. Two other specifications for 0/1 knapsack can be found in Section 3.3.

We can easily construct a memoization algorithm for 0/1 knapsack from specification (g, o) via a template shown in Figure 2. In this template, m is a function that returns a key for an invocation of g. Two invocations have the same key only if their outputs are the same. 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.

Though the template has effectively reused repeated invocations to g, the time complexity of the generated algorithm is still exponential to the number of items. Compared to the standard $O(n^2)$-time algorithm for 0/1 knapsack, such a trivial memoization algorithm is unsatisfactory.

2.2 Thinning and Its Shortages

Before discussing the derivation of efficient memoization algorithms, we first introduce two notations for the convenience of presentation. To distinguish the input and the output of the outermost invocation to g and recursive invocations, we call the input of an arbitrary invocation to g a *search state* (or *state*), as an invocation represents a step in a depth-first search, and call the solution generated by a recursive invocation to g a *partial solution*, as it is not a full solution yet.

The Main Idea of Thinning. The main reason for the ineffectiveness of the trivial memoization algorithm is that the number of partial solutions returned by the generator g can be exponential to the number of items, and memoization keeps this factor unchanged. Therefore, to generate an efficient memoization algorithm, it is important to reduce the number of partial solutions.

Thinning, proposed by Bird and de Moor [1997], is such an approach. As shown in the following program g', to reduce the number of solutions returned by g, thinning inserts a special function *thin* at the return point of g, which prunes off non-optimal solutions from those generated by g.

$$g' = \lambda g'.\lambda xs.thin[R]\ (g\ g'\ xs) \tag{1}$$

Function *thin* is parameterized by a preorder R over the space of solutions. Intuitively, thinning requires preorder R to specify the domination between partial solutions: For any two partial solutions (p_1, p_2), p_1 is worse than p_2 in the sense of R (written as $p_1 R p_2$) only if partial solution p_1 can never lead to the global optimal solution if p_2 exists. Given preorder R and a set of partial solutions P, $thin[R]\ P$ returns one smallest subset of P such that all partial solutions in P are dominated by partial solutions in this subset. According to the requirement to R, $thin[R]$ removes only those non-optimal partial solutions generated by g and does not affect the final result.

Let us take 0/1 knapsack and its specification (g, o) introduced in Figure 1 as an example. In this case, a partial solution is a sub-list of some suffix of the full item list, and it will be completed to a full solution by inserting some (possibly none) items to its front. Therefore, partial solution p_1 is

dominated by p_2 if p_1 not only consumes more capacity but also gains smaller value. At this time, any full solution leaded by p_1 can be improved by replacing items in p_1 with p_2. Such a relation can be described by the following preorder R.

$$p_1 R p_2 \iff (sumw\, p_1 \geq sumw\, p_2) \wedge (sumv\, p_1 \leq sumv\, p_2)$$

There are two noticeable properties of $thin[R]$ that makes g' (Equation 1) efficient.

- First, the number of partial solutions returned by $thin[R]$, which is equal to the number of solutions returned by g', is bounded by the maximum outputs of $sumw$, which is at most W.
- Second, there is an efficient implementation of $thin[R]$ that invokes preorder R only linear times, where the time complexity of R is linear to the size of partial solutions.

Therefore, the time complexity of memoizing (g', o) is only $O(n^3)$, which is exponentially faster than directly memoizing the specification (g, o). For simplicity, we do not go deep into $thin[R]$ in this section. A detailed discussion on *thin* can be find in Section 3.4, which shows that the efficiency of $thin[R]$ is related to the range of *keys* (in this case, $sumw$ and $sumv$) involved in R.

The Shortages of Thinning. So far, we have successfully obtained a polynomial-time memoization algorithm for 0/1 knapsack by applying thinning to a plain specification. However, our previous discussion also exposes two crucial shortages of thinning.

First, thinning requires the user to provide a preorder that specifies the domination between partial solutions. However, finding such a preorder is a non-trivial task and may rely on algorithmic intuitions. In our example, to find a proper preorder for 0/1 knapsack, the user needs to recognize and involve the comparison between the consumed capacity. Actually, the difficulty of finding such a comparison is already close to directly proposing the standard dynamic programming algorithm for 0/1 knapsack, which takes the consumed capacity as the state.

Second, applying thinning solely is not enough to achieve an algorithm that is as efficient as the one proposed by human experts. In our example, there is still a gap between the result of thinning, which runs in $O(n^3)$ time, and the standard $O(n^2)$-time dynamic programming algorithm.

2.3 *MetHyl*: Automating Thinning via Programming-by-Example

For the first shortage, to remove the burden from the user, one natural way is to automate the application of thinning. If a proper preorder can be found automatically, thinning can be treated as a black-box, and thus applying it will not consume any user's effort. In this paper, we propose *MetHyl*, which efficiently synthesizes preorders for thinning via programming-by-example.

Specification for the Preorder. The thinning theorem proposed by Bird and de Moor [1997] provides a formal characterization for the correctness of thinning. *MetHyl* takes this theory as the specification and synthesizes the preorder from it.

The thinning theorem requires the generator to be specified as a (relational) hylomorphism, a common template for recursions in functional programming. In a nutshell, a hylomorphism specification for the generator comprises two separate functions ψ and ϕ.

First, ψ generate a set of *transitions* for a given state, where each transition includes several (possibly none) sub-states for recursion and information used to construct solutions. For example, ψ corresponds to the generator g in Figure 1 may return the following three transitions.

(1) An empty transition when the item list xs is empty.
(2) A direct recursion to state $(tail\, xs)$, representing that the first item is skipped.
(3) A recursion to $(tail\, xs)$ with information $(head\, xs)$, representing that the first item is chosen.

Second, ϕ constructs a set of solutions for each transition and each partial solution of sub-states. For example, the following describes the ϕ corresponding to the generator g in Figure 1.

(1) For an empty transition, ϕ returns an empty list, representing an empty knapsack.

(2) For a direct transition, given a partial solution of the sub-state, ϕ directly returns the partial solution, representing that no item is added to the knapsack.
(3) For the last transition, given an item and partial solution of the sub-state, ϕ adds the item to the partial solution when the capacity is enough, and otherwise returns noting.

Our approaches inherit the requirement on a hylomorphism-style generator from the thinning theorem. We design a language for implementing such programs in Section 3.3 and discuss the effect of this requirement on our approaches in Section 6. For simplicity, we leave the formal definition of hylomorphism to Section 3.2 and still use functions to demo our approaches in this section.

The main advantage of introducing hylomorphism is that how a partial solution is constructed from the partial solutions of sub-states is explicitly specified via function ϕ. Concretely, relation $p \twoheadrightarrow_s p'$, denoting that a partial solution p' of state s can be constructed from another partial solution p, can be extracted from the invocations of ϕ. $p \twoheadrightarrow_s p'$ holds if and only if there is an invocation of ϕ in state s that takes p as the input and generates p'.

The following lists all instances of this relation for 0/1 knapsack specified in Figure 1 when the state xs is $[(3,3),(2,2),(1,2)]$ and the capacity W is 4.

$$\begin{gathered}[] \twoheadrightarrow_{xs} [] \quad [] \twoheadrightarrow_{xs} [(3,3)] \quad [(1,2)] \twoheadrightarrow_{xs} [(1,2)] \quad [(1,2)] \twoheadrightarrow_{xs} [(3,3),(1,2)] \\ [(2,2)] \twoheadrightarrow_{xs} [(2,2)] \quad [(2,2),(1,2)] \twoheadrightarrow_{xs} [(2,2),(1,2)] \end{gathered} \tag{2}$$

In our example, to ensure the correctness, i.e., the equivalence between (g, o) (Figure 1) and (g', o) (Equation 1), the thinning theorem requires preorder R to satisfy the following conditions.

(1) The dominance specified by R is monotonic during the recursion. If partial solution p_1 is worse than p_2 in the sense of R, all partial solutions generated by p_1 must also be worse than those generated by p_2. Concretely, the thinning theorem requires the following condition to hold for any two consecutive states xs and $(tail\ xs)$, any two partial solutions p_1, p_2 of state $(tail\ xs)$, and any partial solution p_1' of state xs such that $p_1 \twoheadrightarrow_{xs} p_1'$.

$$p_1 R p_2 \to \exists p_2' \in (g\ xs), \left(p_2 \twoheadrightarrow_{xs} p_2' \wedge p_1' R p_2'\right) \tag{3}$$

(2) Preorder R implies the order of the objective value. If partial solution p_1 is worse than p_2 in the sense of R, the objective value of p_1 must be no larger than p_2, i.e., $p_1 R p_2 \to (o\ p_1 \le o\ p_2)$.

Intuitively, the first condition ensures that $(g'\ xs)$ is always equivalent to $(thin[R]\ (g\ xs))$, and the second condition ensures that the optimal solution with the largest objective value is always reserved by thinning. Therefore, they together imply the correctness of thinning.

Extracting Examples for the Preorder. The thinning theorem provides a specification, and the remaining task for automating thinning is to find a preorder R satisfying both conditions.

However, directly synthesizing from these conditions is challenging due to the complexity of Formula 3, which involves both universal quantifiers $\forall$ and $\exists$, and a possibly complex relation $\twoheadrightarrow$ defined on the semantics of the input function φ. To our knowledge, there is no efficient synthesizer that can handle such a complex specification.

MetHyl uses the framework of *programming-by-example (PBE)* [Shaw et al. 1975] to resolve this challenge. Given a logic specification, a typical PBE solver first substitutes concrete values into the formula, extracts constraints on *concrete* invocation of the target program (denoted as examples), and then synthesizes from the examples. In this way, the core synthesizer does not have to handle complex logic specifications, and thus the difficulty of synthesis is greatly reduced.

Now, we outline how *MetHyl* extracts simple examples for R from the two conditions.

First, *MetHyl* ensures the second condition by limiting 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 a preorder to be synthesized, and thus considers only Formula 3.

Then, *MetHyl* considering the following formula that is equivalent to Formula 3 for any p_1, p_2.

$$\exists p_1' \in (g\ xs), \left(p_1 \twoheadrightarrow_{xs} p_1' \wedge \forall p_2' \in (g\ xs), \left(\neg p_2 \twoheadrightarrow_{xs} p_2' \vee \neg p_1' R p_2'\right)\right) \rightarrow \neg p_1 R p_2 \tag{4}$$

The conclusion of this formula, $\neg p_1 R p_2$, is extremely simple. If the premise can be transformed to be irrelevant to the unknown preorder R, we can extract example (p_1, p_2) specifying $\neg p_1 R p_2$ by constantly substituting p_1, p_2 with concrete partial solutions until the premise is satisfied. Compared to Formula 3, the constraint provided by this example involves neither universal quantifiers nor $\twoheadrightarrow$, and thus makes it possible to design an efficient synthesizer for R.

The transformation of Formula 4 is motivated by the fixed form of R for ensuring the first condition, which implies that a part of R is known while synthesis. Therefore, we substitute R in the conclusion of Formula 4 with $\lambda p_1.\lambda p_2.p_1 R' p_2 \wedge p_1 ?R p_2$, where R' and $?R$ represent the known preorder and unknown preorder in R respectively, and obtain the following equivalent formula.

$$\left(p_1 R' p_2 \wedge \exists p_1' \in (g\ xs), \left(p_1 \twoheadrightarrow_{xs} p_1' \wedge \forall p_2' \in (g\ xs), \left(\neg p_2 \twoheadrightarrow_{xs} p_2' \vee \neg p_1' R p_2'\right)\right)\right) \rightarrow \neg p_1 ?R p_2$$

Because $\neg p_1' R' p_2$ implies $\neg p_1' R p_2'$, a *weaker* formula whose premise is irrelevant to $?R$ can be obtained by replacing $\neg p_1' R p_2'$ in the premise with the unknown comparison $\neg p_1' R' p_2$.

$$\left(p_1 R' p_2 \wedge \exists p_1' \in (g\ xs), \left(p_1 \twoheadrightarrow_{xs} p_1' \wedge \forall p_2' \in (g\ xs), \left(\neg p_2 \twoheadrightarrow_{xs} p_2' \vee \neg p_1' R' p_2'\right)\right)\right) \rightarrow \neg p_1 ?R p_2 \tag{5}$$

We use an example to show how examples are extracted from Formula 5. In 0/1 knapsack specified by Figure 1, suppose state xs is $[(3,3),(2,2),(1,2)]$, capacity W is 4, and the known part R' is $\lambda p_1.\lambda p_2.sumv\ p_1 \leq sumv\ p_2$. At this time, the domain of p_1, p_2 is $\{[], [(1,2)], [(2,2)], [(2,2),(1,2)]\}$ and the relation $\twoheadrightarrow_{xs}$ is described in Example 2.

- When p_1 and p_2 are taken as $[(1,2)]$ and $[(2,2)]$, the premise of Formula 5 is true because (1) $sumv\ p_1 \leq sumv\ p_2$, (2) $p_1 \twoheadrightarrow_{xs} p_1' = [(3,3),(1,2)]$, and $sumv\ p_1' > sumv\ [(2,2)]$, which is only choice of p_2' satisfying $p_2 \twoheadrightarrow_{xs} p_2'$. Therefore, example $\neg[(1,2)]?R[(2,2)]$ is obtained.
- Similarly, another example can be obtained by taking p_1 and p_2 as $[(1,2)]$ and $[(2,2),(1,2)]$.

Given these two examples, $?R = \lambda p_1.\lambda p_2.sumw\ p_1 \geq sumw\ p_2$ is a valid solution satisfying both examples, which leads to the intended preorder for 0/1 knapsack.

Though Formula 5 is not equivalent to the original specification, i.e., a preorder satisfying Formula 5 may not be valid for Formula 3, the following fact makes it useful for synthesizing R.

- Given preorder R', if no examples for $?R$ can be extracted from Formula 5, or in other word, the premise of Formula 5 is constantly true, R' must be a valid for Formula 3.

This fact suggests an iterative framework for synthesizing R. Starting from $R' = \lambda p_1.\lambda p_2.o\ p_1 \leq o\ p_2$, a preorder satisfying Formula 3 can be synthesized in three steps.

(1) Extract examples for $?R$ from Formula 5 and a set of concrete instances of the COP task.
(2) If no examples are obtained, return $R = R'$ as the synthesis result.
(3) Synthesize a valid preorder $?R$ from the examples, update R' with $\lambda p_1.\lambda p_2.p_1 R' p_2 \wedge p_1 ?R p_2$, and then go back to Step 1.

Synthesizing a Preorder from Example. The remaining task for *MetHyl* 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. Besides, to generate an efficient memoization algorithm, another goal of the synthesis is to minimize the time complexity of the memoization algorithm generated by thinning.

To enable an efficient synthesis algorithm, *MetHyl* synthesizes $?R$ in the following form, which is a conjunction of comparisons related to several key functions.

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

where $?key_i$ is a function mapping a partial solution to an integer and $?op_i$ is an operator in $\{\leq, =, \geq\}$. In our implementation, $?key_i$ is from a grammar including common arithmetic operators and operators for lists and binary trees. More details on this grammar can be found in Section 7.

This form of preorder has two main advantages. First, a preorder in this form can be naturally decomposed to several comparisons $(?op_i, ?key_i)$, where the scale of each comparison is smaller than the preorder. Moreover, in the terms of satisfying the given negative examples, these comparisons are independent of each other: preorder $?R$ satisfies a negative example if and only if some comparison in $?R$ is violated on the example. This property makes it possible to synthesize each comparison separately and thus greatly reduce the scale of the synthesis task.

Second, we prove that the efficiency of the algorithm generated by thinning with a preorder in this form can be estimated by the production of the ranges of all $?key_i$. Such estimation is monotonic while including more comparisons to the preorder, and thus is easy to optimize in a search procedure. More details on this estimation can be found in Section 3.4.

Motivated by both properties, *MetHyl* regards a preorder as a list of comparisons and synthesizes the list incrementally. Starting from an empty list, in each turn, *MetHyl* finds a comparison that is violated on a large enough subset of unsatisfied examples and inserts it to the preorder. The iteration proceeds until all examples are satisfied.

To efficiently find an effective preorder, *MetHyl* makes two changes on this basic iteration.

- As mentioned before, in Section 3.4, we prove that the effectiveness of a preorder for thinning can be estimated by the ranges of the involved key functions. To find an effective preorder, *MetHyl* backtracks on the iteration and uses branch-and-bound, a standard search technique, to optimize the objective function provided by our estimation.
- To restrain the search space, *MetHyl* uses an outermost iteration on two parameters: (1) a size limit n_s for comparisons, and (2) a number limit n_c for comparisons used in the preorder. While choosing the ith comparison for $?R$, only those comparisons that (1) are smaller than n_s, and (2) are violated on at least $1/(n_c - i + 1)$ portions of examples are considered. In this way, the search space of preorders is greatly reduced.

We use an example to show the search procedure of *MetHyl*. Suppose the given negative examples are $([(1,2)], [(2,2)])$ and $([(1,2)], [(1,2),(2,2)])$ extracted in the previous example, and there are only three comparisons $c_1 = (\geq, \lambda p.|p|)$, $c_2 = (\geq, \lambda p.sumw\ p)$ and $c_3 = (\leq, \lambda p.sumv\ p)$ that are smaller than n_s. These three comparisons are violated on 1, 2, and 0 examples respectively, and the ranges[1] of their key functions on these two examples are 2, 3, and 4 respectively. For simplicity, we assume the objective function is exactly the product of ranges of involved key functions.

- When n_c is set to 1, *MetHyl* considers comparisons that are violated on at least $2/(n_c-1+1) = 2$ examples. At this time, c_2 is the only choice and thus *MetHyl* returns $[c_2]$ as the result.
- When n_c is set to 2, *MetHyl* considers comparisons that are violated on at least $2/(n_c-1+1) = 1$ example. At this time, there are two choices c_1 and c_2. First, because $[c_2]$ satisfies all examples, *MetHyl* updates the upper bound to its objective value, which is 3.
  Then, because c_1 is violated only on the second example, *MetHyl* continues to find a preorder satisfying the first example. By branch-and-bound, at this time, *MetHyl* only considers preorders with an objective value smaller than $3/2 = 1.5$. Because there is no comparison with a range smaller than 1.5, *MetHyl* returns immediately and thus takes $[c_2]$ as the result.

[1]The range here is defined as $ma - mi + 1$, where ma and mi are the maximum and the minimum outputs on the examples.

2.4 *MetHyl*$^+$: Improving Thinning via Three Supplementary Rules

For the second shortage, to improve the result of thinning, we analyze the factors that affect the performance of memoizing generator g in Figure 1. The time complexity 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 key function m (introduced in Figure 2) could possibly return, n_{sols} denotes the maximum number of partial 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 memoizing g is the 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 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 our example, we prove that in the general case, with some assumptions, optimizing these four factors is enough to generate an efficient memoization algorithm (Theorem 5.4).

The main shortage of thinning is that it focuses only on n_{sols} while remaining the other three factors unchanged. In our example, though thinning optimizes n_{sols} from $O(2^n)$ to $O(n)$, both s_{st} and s_{sol} remains $O(n)$ in the resulting program and leads to the gap between the result of thinning, which runs in $O(n^3)$-time, and the standard dynamic programming algorithm for 0/1 knapsack.

Motivated by the above analysis, we propose three supplementary rules to optimize the other three factors, and propose solver *MetHyl*$^+$, which automates and integrates these rules into *MetHyl*. For 0/1 knapsack, our rules reduces s_{sol} and s_{st} to $O(1)$, keeps n_{keys} unchanged as $O(n)$, and thus *MetHyl*$^+$ can automatically generate an $O(n^2)$-time memoization algorithm.

The procedure of applying these rules is listed in Figure 3. For each rule, *Initial Program* shows the input for each rule where (g_1, o_1, m_1) is the program generated by thinning, *Intermediate Program* shows the transformation result where red variables represent the unknown functions required by the rule, *Examples* lists concrete examples for the unknown functions extracted from $xs = [(3,3),(2,2),(1,2)]$ and $W = 4$, and *Synthesis Result* shows the functions found by *MetHyl*$^+$.

Rule 1. The first rule optimizes s_{sol}, the size of partial solutions. Note that the list representation of *thinning* is unnecessarily complex in (g_1, o_1, m_1). To run this program, only the weight sum and the value sum of each partial solution matter. Therefore, the main idea of this rule here is to replace the representation of a solution from a list with a more compact representation, which includes only necessary information for calculating the weight sum and the value sum[2].

Rule 1 uses a *converting function* $?f_p$ to convert the representation of partial solutions. It constructs an intermediate program such that each partial solution p generated by the input program is also generated by the intermediate one as the output of $?f_p$. In this procedure, because the type of partial solutions is changed, those functions in the input program that access partial solutions (either takes a solution as an input or constructs a solution as the output) should be replaced correspondingly.

In Figure 3, we mark the four functions that access partial solutions in (g_1, m_1, o_1) as blue, where [] and $\lambda x.\lambda p.(x :: p)$ construct partial solutions, and $\lambda p.(sumw\, p)$ and $\lambda p.(sumv\, p)$ extracts information from partial solutions. Rule 1 replaces them with unknown functions $?c_\perp$, $?c$, $?q_w$ and $?q_v$ respectively and constructs the intermediate program (g_1', o_1', m_1'). In Section 5.1, we show that this transformation can be done by traversing on the AST of the hylomorphism.

[2]Note that after changing the representation, the optimal solution in the original form can still be extracted from the optimized program. In a nutshell, one can trace back the calculation that leads to the optimal solution in the optimized program and recover the optimal solution in the original by repeating the calculation in the input program. This is a standard technique, and thus we omit it throughout this paper.

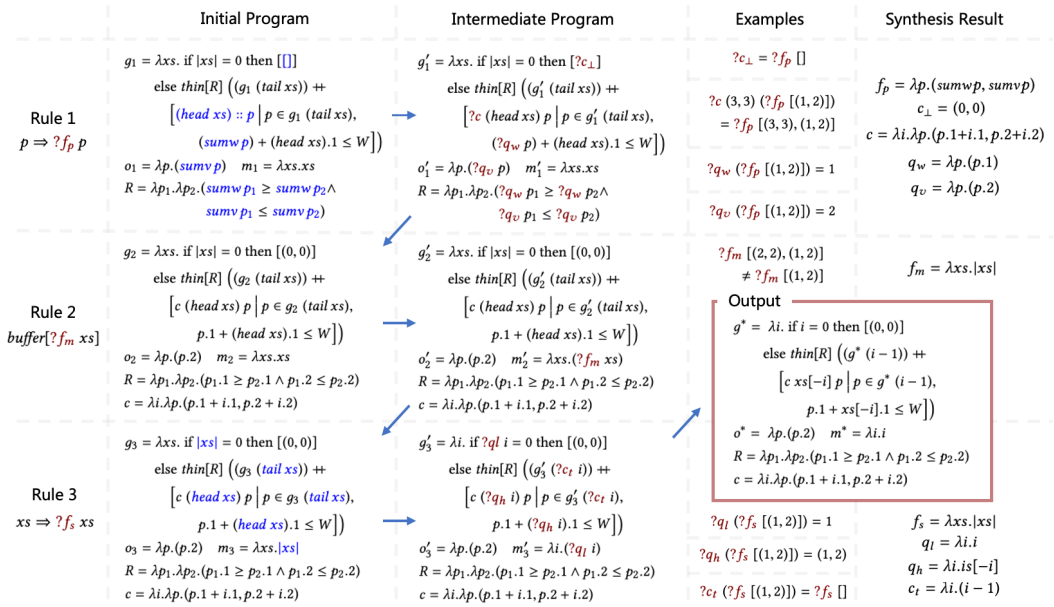

Fig. 3. The procedure of applying the three supplementary rules to improve the result of thinning for 0/1 knapsack. For simplicity, we omit the first parameter of g in this Figure.

MetHyl$^+$ completes the intermediate program by synthesizing functions $?f_p$, $?c_\perp$, $?c$, $?q_w$ and $?q_v$ via programming-by-example. To extract examples for these functions, *MetHyl*$^+$ utilizes the correspondence between the executions of the input program and the intermediate program. Given a concrete instance of 0/1 knapsack, *MetHyl*$^+$ traces the execution of the input program (g_1, o_1, m_1). Each time when a solution-related function is invoked, there must be an invocation of the corresponding unknown function where each involved partial solution p is replaced with the new representation $?f_p\ p$. Such an invocation is recorded as an example for synthesis.

For example, let us consider the invocation of g_1 with state $xs = [(3,3),(2,2),(1,2)]$ and capacity $W = 4$. We highlight two invocation of solution-related functions as follows.

- On partial solution $[(1,2)] \in g_1\ (tail\ xs)$, $\lambda p.(sumv\ p)$ is invoked, and the result is 1. By the correspondence, there should be an invocation of $?q_w$ on the new representation of $[(1,2)]$, i.e., $?f_p\ [(1,2)]$, that outputs 1. Therefore, example $?q_w\ (?f_p\ [(1,2)]) = 1$ is obtained.
- On item $(3,3)$ and partial solution $[(1,2)]$, $\lambda x.\lambda p.(x :: p)$ is invoked, and a new partial solution $[(3,3),(1,2)]$ is constructed. Therefore, according to the correspondence, example $?c\ (3,3)\ (?f_p\ [(1,2)]) = ?f_p\ [(3,3),(1,2)]$ is obtained.

Several extracted examples for the other functions can be found in Figure 3.

The remaining task for applying Rule 1 is to synthesize from the following specification, where E_c, E_w, and E_v are the sets of extracted examples.

$$?c_\perp = ?f_p\ [] \qquad \forall (x,p) \in E_c, ?c\ x\ (?f_p) = ?f_p\ (x :: p)$$
$$\forall p \in E_w, ?q_w\ (?f_p\ p) = sumw\ p \qquad \forall p \in E_v, ?q_v\ (?f_p\ p) = sumv\ p \tag{6}$$

MetHyl$^+$ reduces this task to *lifting problem* and *partial lifting problem*, two kinds of synthesis tasks studied by Ji et al. [2022], and synthesizes by invoking an efficient synthesizer *AutoLifter* for

these tasks. Because the formal definitions of these tasks involve concepts in the category theory, we leave them to Section 3.6 and use two examples to show the reduction made by *MetHyl*$^+$.

- Given functions $q, h, m = \lambda x.\lambda p.(x :: p)$, and a set E of examples, synthesizing $?f$ and $?c$ from the following equation is an instance $\mathsf{LP}(\{m\}, q, h, E)$ of the lifting problem.

$$\forall (x, p) \in E, (q\ (m\ x\ p), ?f\ (m\ x\ p)) = ?c\ x\ (h\ p, ?f\ p)$$

  Clearly, the second formula in Example 6 is equivalent to $\mathsf{LP}(\{\lambda x.\lambda p.(x :: p)\}, \mathit{null}, \mathit{null}, E_c)$, where *null* represents a dummy program returning nothing.
- Given functions $q, h, m = \lambda p.p$, and a set E of examples, synthesizing $?f$ and $?c$ from the following equation is an instance $\mathsf{PLP}(\{m\}, q, h, E)$ of the partial lifting problem.

$$\forall (x, p) \in E, q\ (m\ x\ p) = ?c\ x\ (h\ p, ?f\ p)$$

  Clearly, third formula in Example 6 is equivalent to $\mathsf{PLP}(\{\lambda p.p\}, \lambda p.sumw\ p, \mathit{null}, E_w)$.

Because in the task of applying Rule 1, function $?f_p$ is shared in all specifications, there are some details remaining on merging the results of the reduced tasks, which are left to Section 5.1.

AutoLifter uses grammars to guarantee the efficiency of the synthesis results. Under its default setting, $?f_p$ is synthesized from a grammar including only polynomial-time programs that output only tuples of scalar values, and other functions are synthesized from a grammar including only constant-time operators for scalar values. In this way, the time complexities of all synthesis results except $?f_p$ are guaranteed to be $O(1)$. For Rule 1, because $?f_p$ is never invoked in the optimized program, such a guarantee provided by *AutoLifter* already ensures the efficiency of the result.

Rule 2. The second rule optimizes n_{keys}, the number of keys that m possibly returns. Though the trivial function $m = \lambda xs.xs$ is already efficient in our example, in general, the search state may record too much information such that reusing results only for exactly the same search state is inefficient. An example of this case can be found in Section 3.2, which is shown as Figure 7. To improve this point, Rule 2 replaces the original key function with a more compact one $?f_m$ and thus lets the memoized results be reused between different search states.

To automatically applying Rule 2, *MetHyl*$^+$ synthesizes $?f_m$ from examples. To ensure the correctness of the transformation result (in our example, (g_2', o_2', m_2') in Figure 3), *MetHyl*$^+$ requires $?f_m$ to assign different keys to search states with different outputs. At this time, negative example (a, b) requiring that $?f_m\ a \neq ?f_m\ b$ can be extracted from the execution of the input program.

For example, let us consider the invocation of g_2 in Figure 3 with item list $xs = [(3,3),(2,2),(1,2)]$ and capacity. All invocations of g_2 and their results are listed as follows.

$$g_2\ [] = [(0,0)] \qquad g_2\ [(1,2)] = [(0,0),(1,2)] \qquad g_2\ [(2,2),(1,2)] = [(0,0),(1,2),(3,4)]$$
$$g_2\ [(3,3),(2,2),(1,2)] = [(0,0),(1,2),(3,4),(4,5)]$$

Because the outputs of g_2 are pairwise different on the four states, the outputs of $?f_m$ must be pairwise different on $[]$, $[(1,2)]$, $[(2,2),(1,2)]$, and $[(3,3),(2,2),(1,2)]$, which leads to 6 negative examples in the form of $?f_m\ a \neq ?f_m\ b$ for function $?f_m$.

To focus on effective candidates of $?f_m$, *MetHyl*$^+$ considers only those functions compressed the search state to a tuple of scalar values. At this time, $?f_m$ can be regarded as a tuple of key functions.

$$?f_m\ s = (?key_1\ s, \ldots, ?key_n\ s)$$

MetHyl$^+$ synthesizes $?f_m$ from examples by reducing it to the synthesis task for thinning in *MetHyl*. Note that key function $?f_m$ can be regarded as an equivalence relation $?R_m$ over search states, where $a?R_m b$ is defined as $?f_m\ a = ?f_m\ b$. The following shows the expanded form of $?R_m$.

$$a?R_m b \iff \wedge_i (?key_i\ a) = (?key_i\ b)$$

First, the form of $?R_m$ matches the form of preorders considered by *MetHyl*. Second, the number of different keys returned by $?f_m$ is bounded by the product of the ranges of key functions, which matches the objective function used by *MetHyl*. Therefore, an efficient $?f_m$ can be directly synthesized by invoking the solver in *MetHyl*.

For program (g_2', o_2', m_2'), *MetHyl* synthesizes $?f_m$ as $\lambda xs.|xs|$. Though the value n_{keys} does not change, the synthesized key function simplifies the information on search states required for memorization, and thus make it possible to simplify the representation of search states later.

Rule 3. The step optimizes s_{st}, the size of search states. The procedure of applying this rule is almost the same with Rule 2. First, *MetHyl*$^+$ uses a converting function $?f_s$ to convert the representation of search states, and generates an intermediate program (g_3', o_3', m_3') by replacing all state-related functions (g_3, o_3, m_3) with unknown functions. Second, *MetHyl*$^+$ extracts examples by tracing the execution of (g_3, o_3, m_3) on concrete instances of 0/1 knapsack. Last, *MetHyl*$^+$ synthesizes unknown functions from examples by invoking *AutoLifter*.

For program (g_3', o_3', m_3'), *MetHyl*$^+$ synthesizes $?f_s$ as $\lambda xs.|xs|$ and uses the length of the item list to represent a search state. In this way, s_{st} is reduced from $O(n)$ to $O(1)$.

Result. The result of *MetHyl*$^+$ for 0/1 knapsack is shown as (g^*, o^*, m^*) in Figure 3. In this program, $s_{st} = s_{sol} = O(1)$, $n_{keys} = n_{sols} = O(n)$, and thus the time-complexity is reduced to $O(n^2)$.

3 PRELIMINARIES

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

$$(f_1 \circ f_2)\ x \coloneqq f_1\ (f_2\ x) \quad (f_1 + f_2)\ (i, x) \coloneqq f_i\ x, i \in \{1, 2\}$$
$$(f_1 \times f_2)\ (x, y) \coloneqq (f_1\ x, f_2\ y) \quad (f_1 \triangle f_2)\ x \coloneqq (f_1\ x, f_2\ x)$$

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 set and an arrow from object A to object B is a total function from set A to set B. In a category, a functor F maps objects to objects, arrows to arrows, and keeps 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.

$$\mathsf{I}A \coloneqq A \quad \mathsf{I}f \coloneqq f \quad (!A)B \coloneqq A \quad (!A)f \coloneqq id_A \quad (\mathsf{F}_1 \times \mathsf{F}_2)A \coloneqq \mathsf{F}_1A \times \mathsf{F}_2A \quad (\mathsf{F}_1 \times \mathsf{F}_2)f \coloneqq \mathsf{F}_1f \times \mathsf{F}_2f$$
$$(\mathsf{F}_1 + \mathsf{F}_2)A \coloneqq (\{1\} \times \mathsf{F}_1A) \cup (\{2\} \times \mathsf{F}_2A) \quad (\mathsf{F}_1 + \mathsf{F}_2)f \coloneqq \mathsf{F}_1f + \mathsf{F}_2f$$

where $A \times B$ represents 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* is 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, $cup : \mathsf{PP}A \to \mathsf{P}A$ is defined as $cup\ x \coloneqq \cup_{s \in x} s$, which unions all sets in a set of sets, and $car[\mathsf{F}] : \mathsf{FP}A \to \mathsf{PF}A$ is defined as the following, which is similar to the Cartesian product.

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

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. In the remainder of this paper, we use r^m to denote the result of memoization.

In this paper, recursive generator $rg(\phi, \psi)_\mathsf{F}$ is invoked to generate valid solutions for a given COP. Similar to the discussion in Section 2, we denote the input of $rg(\phi, \psi)_\mathsf{F}$ as a *search state*, an element in the output of $rg(\phi, \psi)_\mathsf{F}$ as a *solution*, and an element in the output of some invocation involved in the recursive definition as *partial solution*.

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 $rg(cup \circ \mathsf{P}\phi, \psi)$.

Compared to a general recursive generator, a hylomorphism assumes the independence while constructing each solution. For hylomorphism $[\![\phi, \psi]\!]_\mathsf{F}$, given the set including all sub-results of the recursions, i.e., $(cup \circ \mathsf{P}(car[\mathsf{F}] \circ \mathsf{F}[\![\phi, \psi]\!]_\mathsf{F}) \circ \psi)$, $[\![\phi, \psi]\!]_\mathsf{F}$ independently constructs solutions for each result via $\mathsf{P}\phi$, and then merges all solutions via *cup*.

Figure 4 shows a program corresponding to the input program (g, o) specified in Figure 1, where g is expressed by hylomorphism $[\![\phi, \psi]\!]_\mathsf{F}$. Because both ϕ, ψ return a set, we use $|collect|$ instead of $|return|$ to express their outputs, where $|collect\ e|$ inserts the value of e to the resulting set.

Relational hylomorphisms are natural for specifying COPs, where ψ and ϕ specify the recursive structure and the construction of solutions respectively. However, relational hylomorphism is not enough to express optimizations (e.g., thinning), where the construction of solutions is no longer independent due to the optimization. Therefore, *MetHyl* takes a relational hylomorphism as the input but expresses the internal optimized programs via recursive generators.

3.3 Programs

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

Definition 3.3 (Equivalence). Two programs (g_1, o_1) and (g_2, o_2) are equivalence on instance i, denoted as $(g_1, o_1) \sim_i (g_2, o_2)$, if $\max(g_1\ p), p \in (g_2\ i) = \max(o_2\ p), p \in (g_2\ i)$.

This concept can be naturally extended to a set of instances. Program (g_1, o_1) and (g_2, o_2) are equivalence on a set I of instances, denoted as $(g_1, o_1) \sim_I (g_2, o_2)$, if $\forall i \in I, (g_1, o_1) \sim_i (g_2, o_2)$.

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 6, and three different programs in $\mathcal{L}_H$ for describing *0/1 knapsack* are shown as Figure 4, 5 and 7, where $|if\ \mathbb{E}\ then\ \mathbb{S}|$ 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 $|\lambda x.\mathbb{S}|$.

$prog = ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$, 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 $(p.1 :: p.2)$;
$o = \lambda p.sumv\ p$

Fig. 4. The program corresponding to (g, o).

$prog_2 = ([\![\phi, \psi]\!]_{\mathsf{F}}, o)$, 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 $p.2$;}
$o = \lambda p.(sumw\ p \leq W)\ ?\ (sumv\ p) : -\infty;$

Fig. 5. A valid program for 0/1 knapsack in $\mathcal{L}_H$. In this program, $-\infty$ is a small enough integer 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; $\mid \mathbb{S}\ \mathbb{S} \mid$ collect $\mathbb{E}$;
		$\mid$	if $\mathbb{E}$ then $\mathbb{S}$ else $\mathbb{S}$
		$\mid$	foreach $x \in [\mathbb{E}, \mathbb{E}]$ in $\mathbb{S}$
Expression	$\mathbb{E}$	$\rightarrow$	$x \mid$ const $\mid \lambda x.\mathbb{E}$
		$\mid$	$\oplus\ \mathbb{E} \ldots \mathbb{E}$

Fig. 6. The syntax of language $\mathcal{L}_H$, where x represents a variable, $\oplus$ 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 $(p.1 :: p.2)$;
$o = \lambda p.sumv\ p$

Fig. 7. A valid program for 0/1 knapsack in $\mathcal{L}_H$. The search state here is (xs, p), where xs is the list of remaining items, p is the list of selected items.

3.4 Thinning

The definition of thinning is based on *preorders*. A preorder on object A is a relation that is reflexive ($\forall a \in A, aRa$) and transitive ($\forall a, b, c \in A, aRb \wedge bRc \rightarrow aRc$).

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

In this paper, we focus on a special case of $thin[R]$ where R is the conjunction of comparisons on several key functions. We denote such a preorder as a *keyword preorder*.

Definition 3.5 (Keyword Preorder). A keyword preorder R of comparisons $\{(op_i, k_i)\}$ on A is defined as $aRb \iff \wedge_i (k_i\ a) op_i (k_i\ b)$, where k_i is an arrow from A to $\mathtt{Int}$ and $op_i \in \{\leq, =, \geq\}$.

Given a keyword preorder R, Theorem 3.6 shows that the size of the set returned by $thin[R]$ and the time cost of $thin[R]$ can both be bounded by the ranges of the key functions involved in R.

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 $\max_1(S)$ returns the largest element in S with default value 1.*

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

- *For any set S, $|thin[R]\ S| \le N_R(S)$.*
- *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 the theorems and move them to the appendix.

3.5 Thinning Theorem

The thinning theorem proposed by Bird and de Moor [1997] shows that $thin[R]$ can be used to derive efficient memoization algorithms for COPs. In this paper, we use the following variant of the thinning theorem, which generalizes the original one to all relational hylomorphisms.

Given program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and a search state s, invoking h on s without memoization generates a search tree where each vertex corresponds to a search state. We introduce two notations T_h and S_h to access the structure of this tree, where $T_h\ s$ and $S_h\ s$ represent the set including all direct children of s and the set including all states in the subtree of s respectively.

Besides, we introduce relation $\twoheadrightarrow_{h,s}$ to denote the constructions of solutions. For partial solution $p \in h\ s$ and tuple $(p_1, \ldots, p_k)$ of partial solutions, $(p_1, \ldots, p_k) \twoheadrightarrow_{h,s} p$ holds if there is an invocation of ϕ where $(p_1, \ldots, p_k)$ are all partial solutions used in the input and p is inside the output.

Theorem 3.7 (Thinning Theorem). *Given program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and preorder R, for any instance i, $(rg(thin[R] \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o) \sim_i (h, o)$ if the following two conditions are satisfied.*

(1) $\forall s \in S_h\ i, \forall p_1, p_2 \in h\ s, p_1 R p_2 \to (o\ p_1 \le o\ p_2)$.

(2) $\forall s \in S_h\ i, \forall \overline{p_1} = (p_{1,1}, \ldots, p_{1,k}), \overline{p_2} = (p_{2,1}, \ldots, p_{2,k})$, where $p_{1,i}$ and $p_{2,i}$ are partial solutions of the same search state for all $i \in [1, k]$, the following formula is always satisfied.

$$\bigwedge_{i=1}^{k} p_{1,i} R p_{2,i} \to \forall p_1', \left(\overline{p_1} \twoheadrightarrow_{h,s} p_1' \to \exists p_2', \left(\overline{p_2} \twoheadrightarrow_{h,s} p_2' \wedge p_1' R p_2'\right)\right) \tag{7}$$

3.6 Lifting and Partial Lifting

Lifting problems and partial lifting problems are synthesis tasks studied by Ji et al. [2022], which are generalized from the synthesis task for automated parallelization.

Definition 3.8. Given arrows p, h starting from object A, a set M including n arrows $m_i : \mathsf{F}_{m_i} A \to A$, and an 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 $?f, ?c_1, \ldots, ?c_n$ such that Equation 8 and 9 are satisfied for all $m_i \in M$, respectively.

$$\forall e \in E[m_i], ((p \triangle ?f) \circ m_i)\ e = (?c_i \circ \mathsf{F}_{m_i}(h \triangle ?f))\ e \tag{8}$$

$$\forall e \in E[m_i], (p \circ m_i)\ e = (?c_i \circ \mathsf{F}_{m_i}(h \triangle ?f))\ e \tag{9}$$

AutoLifter [Ji et al. 2022] 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 *METHYL*: AUTOMATING THINNING

Given program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and a set of instances I, *MetHyl* generates a memoization algorithm by applying thinning to (h, o). Concretely, *MetHyl* synthesizes a keyword preorder $?R$ satisfying Theorem 3.7 for all instance $i \in I$, and returns the following program.

$$prog_1 = (rg(thin[?R] \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}^{m}, o) \tag{10}$$

where m is the trivial key function $\lambda s.s$ for memoization.

4.1 Synthesis Task

To generate an efficient memoization algorithm, *MetHyl* needs to find a keyword preorder $?R$ such that the result of applying thinning with $?R$ is correct and efficient.

For correctness, $?R$ must satisfy the two conditions provided by Theorem 3.7. *MetHyl* ensures the first condition by requiring $?R$ to include comparison $(\leq, o)$, and considers the following equivalent form of Formula 17 in the second condition.

$$\exists p_1', \left(\overline{p_1} \twoheadrightarrow_{h,s} p_1' \wedge \forall p_2', \left(\neg\overline{p_2} \twoheadrightarrow_{h,s} p_2' \vee \neg p_1' R p_2'\right)\right) \rightarrow \bigvee_{i=1}^{k} \neg p_{1,i} R p_{2,i} \tag{11}$$

Given a concrete preorder R and an instance i, we denote pair $(\overline{p_1}, \overline{p_2})$ as a counter-example for R on instance i if the Formula 11 is violated after substituting into $\overline{p_1}$ and $\overline{p_2}$. Let $CE(R, i)$ be the set of all counter-examples of R on instance i. Then finding a correct preorder $?R$ for thinning on instance i is equivalent to finding $?R$ such that $CE(?R, i)$ is empty.

For two keyword preorders R_1 and R_2 where comparisons in R_1 form a subset of those in R_2, Lemma 4.1 relates the sets of counter-examples of R_1 and R_2 with the extra comparisons in R_2.

Lemma 4.1. *Given instance i, for any two keyword preorders R_1, R_2 where all comparisons in R_1 are included in R_2, the following formula is always satisfied.*

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

where R_2/R_1 represents the keyword preorder formed by the comparisons in R_2 that are not used in R_1.

Lemma 4.1 suggests an incremental synthesis scheme for $?R$. To synthesize a correct preorder by enlarging a known keyword preorder R_1 with an unknown one $?R_2$, $?R_2$ must *satisfy* all examples $(\overline{p_1}, \overline{p_2})$ in $CE(R_1, i)$, where $?R_2$ satisfies example $(\overline{p_1}, \overline{p_2})$ is defined as $\neg\overline{p_1}?R_2\overline{p_2}$.

For efficiency, the number of plans returned by $thin[?R]$ and the time cost should be minimized. Guided by Theorem 3.6, *MetHyl* optimizes the following objective function while synthesis.

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

4.2 Synthesis Algorithm

We start with a synthesis algorithm for a subtask where a finite set of comparisons C and a size limit n_c are provided. In this subtask, the search space of $?R$ is constrained to keyword preorders constructed by $(\leq, o)$ and at most n_c comparisons in C.

As shown in Algorithm 1, *MetHyl* solves this subtask via *branch-and-bound*. The main function `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 cost (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).

The implementation of `CandidateComps()` is crucial to the efficiency of Algorithm 1. If it returns too many candidate comparisons, the search space of `BestPreorder` will be too large to be explored efficiently. Based on the following lemma, `CandidateComps`(C, lim, es) in *MetHyl* returns only those comparisons that satisfy at least $|es|/lim$ examples.

Lemma 4.2. *Given a set of instances I, for any two keyword preorders R_1, R_2 where all comparisons in R_1 are included in R_2 and $\forall i \in I, CE(R_2, i) = \emptyset$, 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.,*

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

Algorithm 1: Synthesizing preorder $?R$ for applying thinning.

Input: Input program (h, o), a set of instances I, a set of comparisons $C = \{(op_i, k_i)\}$, and a size limit n_c.
Output: A keyword preorder $?R$ for (h, o) that involves only comparisons in C.

```
1  Function BestPreorder(R, lim, costLim):
2      es ← ∪_{i∈I} CE(R, i);
3      if es = ∅ then return R;
4      if lim = 0 then return ⊥;
5      cList ← CandidateComps(C, lim, es);
6      Sort cList in the increasing order of cost(R ∪ {c}, I);
7      res ← ⊥;
8      foreach c ∈ cList do
9          if cost(R ∪ {c}, I) ≥ costLim then continue;
10         R′ ← BestPreorder(R ∪ {c}, lim − 1, costLim);
11         if R′ ≠ ⊥ then (res, costLim) ← (R′, cost(R′, I));
12     end
13     return res;
14 return BestPreorder({(≤, o)}, n_c, +∞);
```

where $|R|$ represents the number of comparisons in keyword preorder R.

A direct implementation of `CandidateComps`(C, lim, es) in Algorithm 1 (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* further improves this point by sampling. For each comparison in C, *MetHyl* decides whether to include it in `CandidateComps`(C, lim, es) in two steps.

- First, *MetHyl* draws $lim \times n_t$ random examples from es, where n_t is a given parameter. The comparison will be ignored if the number of samples it satisfies is smaller than $k_t = \lfloor lim/2 \rfloor$.
- Second, the comparison is evaluated on all examples and is returned by `CandidateComps` only when it satisfies at least $|es|/lim$ examples.

Specially, k_t is set to $lim \times n_t$, i.e., the number of samples, when lim is equal to 1.

By Chernoff bound, the probability for a comparison that satisfies at least $1/lim$ portion of examples in es to be filtered out by the samples is at most $\exp(-n_t/8)$. Therefore, the probability for Algorithm 1 to incorrectly exclude some comparison can be controlled by n_t. Moreover, as we shall show later, because Algorithm 1 in *MetHyl* is wrapped in an iteration procedure, such an error rate does not affect the completeness of *MetHyl*, as demonstrated by Theorem 4.3.

The following is some details to make Algorithm 1 useful for synthesizing $?R$ in practice.

Decide a finite set of comparisons. In practice, the space of comparisons is specified by a grammar, in which the number of comparisons is infinite. Because of the difficulty of finding the optimal preorder from an infinite set of comparisons, *MetHyl* approximates it via the principle of *Occam's Razor* and prefers to use smaller comparisons to construct $?R$. Concretely, *MetHyl* selects a parameter s_c and takes all comparisons no larger than s_c in the grammar as set C.

Decide s_c and n_c. Because both parameters s_c (the size limit of comparisons) and n_c (the number limit of comparisons) are not given in practice, *MetHyl* decides them iteratively with two parameters s_c^* and n_c^*. In each turn, *MetHyl* considers the subtask where $n_c = n_c^*$ and $s_c = s_c^*$. If there is no solution, both s_c^* and n_c^* will be increased by 1 in the next iteration.

We prove the completeness of *MetHyl* for synthesizing preorders in the following theorem.

Theorem 4.3. *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) $\forall i \in I, CE(R, i) = \emptyset$, and (2) R is constructed by $(\leq, o)$ and some comparisons in G, MetHyl must terminate and return such a keyword preorder.*

5 *MetHyl*$^+$: IMPROVING THINNING VIA THREE RULES

To improve the memoization algorithm generated by thinning, *MetHyl*$^+$ introduce three supplementary rules on the basis of *MetHyl* to optimize three factors ignored by thinning: (1) the size of solutions, (2) the number of keys in memoization, and (3) the size of search states.

5.1 Rule 1: Optimizing the Representation of Solutions

The program $prog_1$ generated by thinning is in Form 10, where the bodies of ϕ and ψ are statements ($\mathbb{S}$) in language $\mathcal{L}_H$ (Figure 6). *MetHyl*$^+$ optimizes the size of partial solutions via a converting function $?f_p$, which maps partial solutions into a tuple of scalar values.

MetHyl$^+$ constructs an intermediate program $prog_1'$ that performs almost the same as $prog_1$ except each partial solution p is stored as $?f_p\ p$. The construction is done by rewriting all solution-related functions, which can be classified into the following two types according to their output type.

- Constructors, which constructs a new partial solution. In language $\mathcal{L}_H$, constructors are the sub-expressions sp_e of all $|collect\ sp_e|$ in ϕ.
- Queries, which returns a tuple of scalar values. In language $\mathcal{L}_H$, queries are all solution-related expressions outside $|collect\ sp_e|$ in program $prog_1$.

The following shows the content of $prog_1'$.

$$prog_1' = (rg((thin[R']) \circ cup \circ \mathsf{P}\phi', \psi)_{\mathsf{F}}, ?q[o]) \tag{12}$$

First, Both the objective function o and keywords $?R$ are queries. In $prog_1'$, o and $?R$ are replaced with $?q[o]$ and $R' = \{(op, ?q[k]) \mid (op, k) \in R\}$. Second, constructors and other queries are extracted from ϕ. In $prog_1'$, ϕ is replaced with function ϕ' constructed by Algorithm 2, a structural recursion on the AST of ϕ. The notations used in Algorithm 2 are explained blow.

- $|s|$ represents 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 $prog_1$ 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 $RE(p, i)$.

- For each query (or constructor) p, functor $\mathsf{F}[p]$ indicates partial solutions in the input of p. For queries extracted from $R?$ and o, $\mathsf{F}[p] := \mathsf{I}$; For functions extracted from ϕ, $\mathsf{F}[p] := \mathsf{F}\times!T_1 \times \ldots !T_n$, where F is the functor used by the recursive generator in $prog_1$, and $!T_i$ is the type of the ith temporary variable used by p[3].
- Given instance i, for each query (or constructor) p, set $RE(p, i)$ records all the inputs on which p is invoked during $(prog_1\ i)$, which can be obtained by instrumenting $prog_1$.

Lemma 5.1 provides a sufficient condition for $?f_p$, $?q$ and $?c$ to guarantee the correctness of $prog_1'$.

[3]Note that there are only two ways to introduce a temporary variable in language $\mathcal{L}_H$, which are lambda expressions $(\lambda x.\mathbb{E})$ and for-loops (foreach $x \in [\mathbb{E}, \mathbb{E}]$ in $\mathbb{S}$).

Algorithm 2: Construct ϕ' from ϕ.

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

Lemma 5.1. *Given instance i and program $prog_1$ in Form 10, let $prog'_1$ be result of Rule 1. If for any query q and constructor m, Formula 13 and Formula 14 are satisfied respectively, $prog_1 \sim_i prog'_1$ holds.*

$$\forall e \in RE(q, i), q\ e = ?q[q]\ (\mathsf{F}[q]?f_p\ e) \tag{13}$$

$$\forall e \in RE(m, i), ?f_p\ (m\ e) = ?c[m]\ (\mathsf{F}[m]?f_p\ e) \tag{14}$$

To synthesize from Formula 13 and 14, *MetHyl*$^+$ first rewrite Formula 13 as the follows.

$$(q \circ id)\ e = (?q[q] \circ \mathsf{F}[q](null \triangle ?f_p))\ e \tag{15}$$

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

Compared to Definition 3.8, Formula 15 is equal to the specification of partial lifting problem $\mathsf{PLP}(\{id\}, q, null, \cup RE(q, i))$. Therefore, *MetHyl*$^+$ invokes *AutoLifter* to solve this task and find a solution $?q[q] = q[q]$ and $?f_p = f_p[q]$ for each query q.

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 13 by adjusting the input type of $q[q]$.

Next, *MetHyl*$^+$ substitutes $?f_p$ into the Formula 14 and rewrites the result into the follows.

$$\forall m \in M, ((f_q \triangle ?f'_p) \circ m)\ e = (?c[m] \circ \mathsf{F}[m](f_q \triangle ?f_p))\ e \tag{16}$$

where M represents the set of constructors. Compared to Definition 3.8 gain, Formula 16 is equal to 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 all $?c[m]$ can be synthesized by invoking *AutoLifter* again.

At last, *MetHyl* fills the synthesized $?f_p, ?q[q]$ and $?c[m]$ into the intermediate program $prog'_1$ and thus obtains the result of applying Rule 1.

5.2 Rule 2: Optimizing the Number of Keys

Given program (r, o), where r is a recursive generator, and a set of instances I, *MetHyl*$^+$ optimizes the number of keys by synthesizing a proper key function $?f_m$ and returns $(r^{?f_m}, o)$ as the result.

Lemma 5.2 provides a sufficient condition for $?f_m$ to guarantee the correctness of $(r^{?f_m}, o)$.

Lemma 5.2. *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 $s_1, s_2 \in (S_r\ i)$, $r\ s_1 \neq r\ s_2 \rightarrow ?f_m\ s_1 \neq ?f_m\ s_2$.*

Given an instance i, a set of examples $ME(i)$ can be extracted according to Lemma 5.2. Each example in $ME(i)$ is a pair of states, on which $?f_m$ must output different keys.

For efficiency, to limit the number of keys, *MetHyl*$^+$ requires $?f_m$ to return a tuple of scalar values, and thus synthesizes it in the form of $\lambda s.(?key_1\ s, \ldots, ?key_n\ s)$. At this time, the number of keys returned by $?f_m$ is bounded by the product of the ranges of all $?key_i$. Therefore, *MetHyl*$^+$ minimizes the following objective function while synthesizing $?f_m$.

$$cost(?f_m, I) \coloneqq \prod_{i=1}^{n} range(k_i, S), \text{ where } S = \cup_{i \in I}(S_r\ i)$$

Notice that $?f_m$ can be regraded as a preorder $?R_m$ where $s_1 ?R_m s_2$ is defined as $\wedge_i(key_i\ a) = (key_i\ b)$, and $cost(?f_m, I)$ is exactly the same as $N_{?R_m}(\cup_{i \in I}, S_r\ i)$ defined in Theorem 3.6. Therefore, the synthesis task of $?f_m$ has the same form as the synthesis task for thinning discussed in Section 4.1, and can be solved by the synthesis algorithm proposed in Section 4.2.

5.3 Rule 3: Optimizing the Representation of Search States

After applying the second rule, the program is transformed into the following form.

$$prog_3 = (r, o), \quad r = rg\Big(thin[R] \circ cup \circ \mathsf{P}\phi, \psi\Big)_{\mathsf{F}}^{f_m}$$

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

Similar to Rule 1, *MetHyl*$^+$ optimizes the size of states via a converting function $?f_s$, which maps states in $prog_3$ into a tuple of scalar values. *MetHyl*$^+$ rewrites all state-related queries and constructors in $prog_3$ with functions $?q$ and $?c$ respectively, and thus constructs an intermediate program $prog'_3$ that performs almost the same as $prog_3$ except state s in $prog_3$ is stored as $?f_s\ s$.

In $prog_3$, key function f_m is identified as a query. Other queries and constructors are extracted from ψ similarly to Algorithm 2. The only difference is that in ψ, the parameter sp_e of $|collect\ sp_e|$ is a structure including states, on which r will be recursively applied, and scalar 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., $|sp_e.i_1.i_2 \ldots i_n|$. Those components corresponding to states and scalar values are identified as constructors and queries respectively.

We omit the concrete synthesis task and the synthesis algorithm here because they are the same as the counterparts for Rule 1.

5.4 Properties of *MetHyl*$^+$

We end this section with two noticeable 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 generated by MetHyl$^+$ with I. Then $\forall i \in I$, $(h, o) \sim_i p^*$.*

Second, Theorem 5.4 shows when ϕ, ψ and o in the input program and all functions involved in the program space run in pseudo-polynomial time and involve only linear arithmetic operators, the program synthesized by *MetHyl*$^+$ is guaranteed to run in pseudo-polynomial time, i.e., the time complexity of the result is polynomial to the size and values in the input.

Theorem 5.4. *Given input* $([\![\phi, \psi]\!]_F, o)$ *and grammar* G *specifying the program space for synthesis tasks, the program generated by MetHyl*$^+$ *must be pseudo-polynomial time if the following conditions 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 DISCUSSION

In this section, we discuss three subtle details on the design of *MetHyl* and *MetHyl*$^+$.

The order of transformations. *MetHyl*$^+$ optimizes the input program by applying thinning, Rule 1, Rule 2, and Rule 3 in order. Such an order is decided because of the following reasons.

First, Rule 1 should be applied after thinning. Because Rule 1 ignores most information in the solution, some attributes may become incalculable after applying Rule 1. In the example discussed in Section 2, after replacing solution p with $(sumw\ p, sumv\ p)$, the size $|p|$ becomes incalculable in the optimized program. Therefore, if Rule 1 is applied before thinning, some efficient preorders for thinning may become incalculable and thus an inefficient preorder may be used.

Second, Rule 3 should be applied after Rule 2. Before applying Rule 2, the key function used for memoization is $\lambda s.s$, which means that the whole state s is necessary. Therefore, applying Rule 3 before Rule 2 cannot lead to any optimization.

Third, thinning and Rule 1 should be applied before Rule 2. Because both thinning and Rule 1 simplify the output of the generator, applying them before Rule 2 may let the generator output the same on more states and thus may enable a more efficient key function.

The requirement on the relational hylomorphism. Following the thinning theorem (Theorem 3.7), *MetHyl* requires the generator in the input program to be specified in the form of relational hylomorphism. We believe this requirement is not a significant limitation in practice because of the following two reasons.

First, as shown in Section 2.3, a generator can be converted to hylomorphism by specifying the recursion of states and the construction of solutions separately. Such a conversion does not take algorithmic effort and should be easier for the user than proposing an efficient algorithm.

Second, there have been studies on automatically generating a hylomorphism from a recursive program [Hu et al. 1996]. Therefore, this limitation can be eliminated by combining *MetHyl* and *MetHyl*$^+$ with these approaches.

The correctness guarantee of the result. Both *MetHyl* and *MetHyl*$^+$ ensure the correctness of the result only on a given set of instances. Such a guarantee can be improved to the correctness on all instances via a complete verifier and the CEGIS framework [Solar-Lezama et al. 2006].

However, due to the complexity of memoization, such a verifier may not exist. Therefore, in our implementation we use the probabilistic verifier provided by *AutoLifter*, which verifies quickly by testing the result on a dynamically adjusted number of random instances. The guarantee provided by this verifier follows the framework of PAC learnability [Valiant 1984], which ensures the probability for the error rate to be larger than a threshold is small. The practical performance of such guarantees has been demonstrated by Ji et al. [2021]; Wang et al. [2021].

7 IMPLEMENTATION

Our implementation of *MetHyl* and *MetHyl*$^+$ can be found in the supplementary material.

Generating instances. *MetHyl* and *MetHyl*$^+$ require a set of instances to extract examples for each synthesis task. We generate these instances by sampling. Given an instance space specifying by assigning each integer with a range and each recursive data structure with an upperbound on the size, *MetHyl* and *MetHyl*$^+$ sample from the space according to a uniform distribution.

To limit the time cost of executing the input program, *MetHyl* and *MetHyl*$^+$ decide this space by iteratively squeezing from a default space until the average time cost of the input program on a random instance is smaller than 10^{-3} second.

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 applying Rule 3, as it can correspond the search state to some global data structure.

Others. The original implementation of *AutoLifter* uses *PolyGen* [Ji et al. 2021], a specialized solver for conditional linear expressions, to synthesize ?*c* for lifting problems. Therefore, 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 4.2, 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` to be at most $e^{-1} \approx 37\%$.

To use the iterative verifier discussed in Section 6, we set the basic number of examples to 10^4 and thus the probability for the error rate of the result to be more than 10^{-3} is at most 1.82×10^{-4}.

8 EVALUATION

Our evaluation answers two research questions.

- **RQ1**: How is the overall performance of *MetHyl* and *MetHyl*$^+$?
- **RQ2**: How do thinning and the three rules proposed in this paper perform in *MetHyl*$^+$?

8.1 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.

The input program is written in our language $\mathcal{L}_H$ defined in Section 3.3, where each program comprises a generator in the form of relational hylomorphism and a scorer. We divide the 17 COPs into two categories according to whether solutions are constrained, and construct input programs for them via two criteria respectively.

The first category includes COPs where solutions are not constrained. A representative COP here is *rod cutting*, the first example in the 15th chapter of *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 of all pieces.*

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

The second category includes COPs where solutions are constrained. A representative COP in this category 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.

- The first program keeps all information in the input of ψ and filters out invalid solutions while deciding transitions. Figure 7 shows such a program for *0/1 knapsack*.
- The second program tries all possibilities of constructing solutions in ψ and lets ϕ filter out invalid ones. Figure 4 shows such a program for *0/1 knapsack*.
- The last program uses g to generate all solutions and excludes invalid ones via a small enough objective value. Figure 5 shows such a program for *0/1 knapsack*.

8.2 Experiment Setup

We run *MetHyl* and *MetHyl*$^+$ on all 37 tasks in the dataset and set a time limit of 120 seconds for *MetHyl* and each rule in *MetHyl*$^+$. Especially, if a rule in *MetHyl*$^+$ times out, *MetHyl*$^+$ will skip this rule and go on to the next.

For each execution, we record the time cost and the transformation result for *MetHyl* and each rule in *MetHyl*$^+$. We manually verify the correctness and the time complexity of each result and compare them with the reference algorithms provided by Cormen et al. [2009] and Li [2011].

8.3 RQ1: Overall Performance of *MetHyl* and *MetHyl*$^+$

We confirm that all programs generated by *MetHyl* and *MetHyl*$^+$ are completely correct. The detailed performance of *MetHyl* and *MetHyl*$^+$ are listed in Table 1.

- For each task, *COP* lists the name of the corresponding COP in *Introduction to Algorithm*, *Imp* lists the index of the principle used to implement the input program if the COP is in the second category, and T_{inp} lists the time complexity of the input program in $\tilde{\Omega}$ notation.
- For each approach, T_{res} lists the time complexity of the result program in $\tilde{O}$ notation, and *Time* lists the number of seconds used to generate the result.

First, Table 1 demonstrates the effectiveness of *MetHyl* on automating thinning. *MetHyl* achieves exponential speed-ups against the input program on 31/37(83.8%) tasks with an average time cost of 4.2 seconds. Besides, the comparison between the results of *MetHyl* and the reference algorithms demonstrates the gap between thinning and human experts: *MetHyl* achieves the same time complexity as the reference algorithm only on 3/37(8.1%) tasks.

Second, Table 1 demonstrates the overall effectiveness of *MetHyl*$^+$ on synthesizing efficient memoization algorithms. *MetHyl*$^+$ achieves exponential speed-ups against the input program on 36/37(97.3%) tasks with an average time cost of 59.2 seconds. Moreover, compared to *MetHyl*, the ability of *MetHyl*$^+$ is much closer to human experts: *MetHyl*$^+$ achieves the same time complexity as the reference algorithm on 26/37(70.3%) tasks.

Besides, we conduct a case study on those 11 tasks where *MetHyl*$^+$ fails in achieving the same complexity as the reference algorithms, and conclude the following three main 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 $-\infty$ according to the third principle. At this time, *MetHyl*$^+$ can hardly extract examples for $?R$ and thus fails in applying thinning.
- 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 all the three supplementary rules in *MetHyl*$^+$ keep the recursive structure unchanged,

Table 1. The performance of *MetHyl* and *MetHyl*$^+$ on all tasks in our dataset.

COP	Imp	T_{inp}	*MetHyl*		*MetHyl*$^+$	
			T_{res}	Time	T_{res}	Time
0/1 Knapsack	1	2^n	2^n	0.1	n^2†	25.9
	2		n^3	1.8		11.9
	3			3.8		23.8
Rod Cutting		2^n	n^3	0.1	n^2†	18.4
Matrix Chain		4^n	n^4	0.1	n^3†	15.4
LCS	1	5.8^n	n^3	0.1	n^2†	6.9
	2			0.1		5.9
	3			0.3		24.3
Optimal BST		4^n	n^4	0.2	n^3†	24.5
15-1		2^n	n^4	0.1	n^4	123.7
15-2	1	2^n	n^3	0.1	n^2†	8.7
	2			0.1		5.6
	3			0.3		20.9
15-3		2^n	n^3	1.5	n^3	133.4
15-4	1	2^n	n^3	0.1	n^3	141.4
	2			0.2		240.5
	3			0.3		240.6
15-5	1	7.6^n	n^3	0.1	n^2†	40.1
	2			0.1		41.3
	3	$\approx 7.7^n$‡		120.0	7.7^n	274.9

COP	Imp	T_{inp}	*MetHyl*		*MetHyl*$^+$	
			T_{res}	Time	T_{res}	Time
15-6	1	2^n	2^n	0.5	n†	41.2
	2		n^2	0.4		28.1
	3			0.8		31.4
15-7	1	n^n	n^3†	0.1	n^3†	3.6
	2			0.1		10.5
	3			0.2		8.9
15-8	1	2^n	2^n	0.1	n^2†	31.3
	2	n^n	n^4	0.2	n^3	25.5
	3			0.2		12.1
15-9		4^n	n^4	0.1	n^3†	4.8
15-10		n^n	n^4	0.1	n^4	34.1
15-11	1	n^n	n^n	0.1	n^3†	142.0
	2		n^4	9.0	n^4	137.5
	3			9.9		137.1
15-12	1	n^n	n^n	0.1	n^3†	54.9
	2		n^5	2.1		27.6
	3			2.9		31.6

† The result achieve the same time complexity as the reference algorithm for the corresponding COP.
‡ Both time complexities 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.

Name	Exp	Poly	⊥	Time	Name	Exp	Poly	⊥	Time
Thinning	31	0	1	4.2	Rule 1	0	3	7	32.8
Rule 2	5	0	0	5.1	Rule 3	0	24	3	59.2

this non-optimal behavior remains in the result and thus leads to a higher time complexity. Optimizing the recursive structure of hylomorphism is future work.

- For the other 8 tasks, *MetHyl*$^+$ fails because their input programs involve complex non-linear expressions. As mentioned in Section 7, *MetHyl*$^+$ uses an enumerative solver, namely *observational equivalence*, to synthesize ?*c* required by the first and the third supplementary rules when non-linear expressions are involved. Because the scalability of this solver is limited, *MetHyl*$^+$ times out while applying the first and the third rules when the target ?*c* is non-linear and too large, e.g., $x_1 - \text{dis}(p_1, p_2) + \text{dis}(p_1, p_3) + \text{dis}(p_2, p_3)$ in Task 15-3, where $\text{dis}(p, q)$ 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*.

8.4 RQ2: Performance of Thinning and Supplementary Rules in *MetHyl*$^+$

We manually analyze all intermediate results of *MetHyl*$^+$ and summarize the performance of thinning and each supplementary rule in Table 2, where *Name* lists the name of the corresponding

rule, *Exp* lists the number of tasks where the rule achieves an exponential speed-up, *Poly* lists the number of tasks where the rule 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, thinning and the second rule produce all exponential speed-ups, while the first rule and the third rule produce all polynomial ones. This result matches the target of each rule: (1) the number of solutions, optimized by thinning, and the number states, optimized by the second rule, can be exponential, and (2) the scale of solutions, optimized by the first rule, and the scale of the states, optimized by the third rule, are usually polynomial.

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

9 RELATED WORK

Program Calculation. This paper 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] manually derive dynamic programming algorithms by *thinning*. Among them, Morihata et al. [2014] notice that applying thinning solely may not be enough to derive an efficient dynamic programming algorithm, and uses a rule namely *incrementalization* to optimize the program generated by thinning. Compared to the supplementary rules proposed in our paper, the application of this rule is still manual and is restricted to associative and commutative operators.

Second, there are several existing studies on automating thinning. Bird [2001]; Sasano et al. [2000] focus only on a special kind of COPs namely *maximum marking problem*, where a partial solution is a set of weighted items and the objective function is the total weight of the selected items. Morihata [2011] focuses on a special kind of hylomorphisms namely sequential decision procedures, which recurses strictly according to the structure of a list. In comparison, the scopes of these three approaches are strictly more narrow than our approaches *MetHyl* and *MetHyl*$^+$. Only 9/37(24.3%) tasks in our dataset are instances of the maximum marking problem, and only 10/37(27.0%) tasks in our dataset use sequential decision procedures.

Last, there are other approaches for deriving dynamic programming beside thinning [Giegerich et al. 2004; Lin et al. 2021; Liu and Stoller 2003; 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], of which the scope is strictly more narrow than ours due to the requirement on 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 synthesizing dynamic programming algorithms.

Our approaches use program synthesizers to (1) synthesize preorders for thinning and Rule 2, and (2) synthesize program fragments for Rule 1 and Rule 3. The specification of both tasks are instances of relational specification, and thus our approach is related to *Relish* [Wang et al. 2018b], a general solver for relational specifications. However, *Relish* cannot be applied to our tasks because (1) *Relish* cannot optimize an objective function while synthesis, and (2) the *finite tree automata* used by *Relish* does not directly support lambda expressions, which is included in our grammar.

There are also many 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 in both synthesis tasks, and thus all these synthesizers are unavailable.

10 CONCLUSION

In this paper, we propose two novel synthesizers *MetHyl* and *MetHyl*$^+$ for automatically synthesizing efficient memoization for relational hylomorphism. We demonstrate the efficiency and effectiveness of both approaches on a dataset including 37 tasks related to 17 COPs in our evaluation.

This paper is motivated by the theories in program calculation for deriving 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 our approaches 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 S. Bird. 2001. Maximum marking problems. *J. Funct. Program.* 11, 4 (2001), 411–424. https://doi.org/10.1017/S0956796801004038

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. 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

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. 2021. 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. 2022. Black-Box Algorithm Synthesis – Divide-and-Conquer and More. arXiv:2202.12193 [cs.PL]

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

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.

Akimasa Morihata. 2011. A Short Cut to Optimal Sequences. *New Gener. Comput.* 29, 1 (2011), 31–59. https://doi.org/10.1007/s00354-010-0098-4

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

Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. 2000. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. *ACM SIGPLAN Notices* 35, 9 (2000), 137–149.

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.

David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*.

260–267. http://ijcai.org/Proceedings/75/Papers/037.pdf

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 326–340. https://doi.org/10.1145/2908080.2908102

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 287–296. https://doi.org/10.1145/2491956.2462174

Leslie G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142. https://doi.org/10.1145/1968.1972

Bo Wang, Teodora Baluta, Aashish Kolluri, and Prateek Saxena. 2021. SynGuar: guaranteeing generalization in programming by example. 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, 677–689. https://doi.org/10.1145/3468264.3468621

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.

A APPENDIX

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

Theorem A.1 (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 $\max_1(S)$ returns the largest element in S with default value 1.*

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

- *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 $K = \{k_1, \ldots, k_m\}$ 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 $k_1 \triangle \ldots \triangle k_m$. Then $N_R(S)$ is the range of f_k. By the definition of the keyword preorder, we have the following formula:

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

In other words, for elements where the outputs of the feature function are the same, their order in R is total. Therefore, the number of maximal values in S is no more than the range of the key function, 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 $\arg\max_a k^*\ a, a \in S \wedge wR_x(f_k\ a)$. 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.

To prove this claim, we make an induction on 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 $w_i.x = i$.

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 $f_k = k_1 \triangle \ldots \triangle k_m$ 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 $\mathbb{W} \leftarrow [mi_1, ma_1] \times [mi_2, ma_2] \times \cdots \times [mi_m, ma_m]$;
5 $\forall w \in \mathbb{W}, Val[w] \leftarrow \bot$;
6 **foreach** $a \in S$ **do**
7 **if** $Val[f_k\ a] = \bot \vee k^*\ (Val[f_k\ a]) \leq k^*\ a$ **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** $w.i = mi_i \vee Val[w] = \bot$ **then continue**;
15 $w' \leftarrow w$; $w'.i \leftarrow w.i - 1$;
16 **if** $Val[w'] = \bot \vee k^*\ Val[w'] \leq k^*\ Val[w]$ **then**
17 $Val[w'] \leftarrow Val[w]$;
18 **end**
19 **end**
20 **end**
21 **return** $\{a \mid a \in S \wedge Val[f_k\ a] = a\}$;

According to Lines 11-20 in Algorithm 3, $Val[w]$ is equal to the element with the largest output of k^* among $Val'[w_{w.i}], Val'[w_{w.i+1}], \ldots, Val'[w_{ma_x}]$. (For simplicity, we define $k^*\ \bot$ 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.

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

This fact contradicts with the inductive hypothesis and thus the induction holds. □

Theorem A.2 (Theorem 3.7). *Given program $(h = [\![\phi, \psi]\!]_{\mathsf{F}}, o)$ and preorder R, for any instance i, $(rg(thin[R] \circ cup \circ \mathrm{P}\phi, \psi)_{\mathsf{F}}, o) \sim_i (h, o)$ if the following two conditions are satisfied.*

(1) $\forall s \in S_h\ i, \forall p_1, p_2 \in h\ s, p_1 R p_2 \rightarrow (o\ p_1 \leq o\ p_2)$.
(2) $\forall s \in S_h\ i, \forall \overline{p_1} = (p_{1,1}, \ldots, p_{1,k}), \overline{p_2} = (p_{2,1}, \ldots, p_{2,k})$, where $p_{1,i}$ and $p_{2,i}$ are partial solutions of the same search state for all $i \in [1, k]$, the following formula is always satisfied.

$$\bigwedge_{i=1}^{k} p_{1,i} R p_{2,i} \rightarrow \forall p_1', \left(\overline{p_1} \twoheadrightarrow_{h,s} p_1' \rightarrow \exists p_2', \left(\overline{p_2} \twoheadrightarrow_{h,s} p_2' \wedge p_1' R p_2'\right)\right) \tag{17}$$

Proof. For simplicity, we use r to denote $rg(thin[R] \circ cup \circ \mathrm{P}\phi, \psi)_{\mathsf{F}}$. Because ψ in h is also used in r, the search tree 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.

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

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

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

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 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, the remaining task is to prove the second claim via the inductive hypothesis. Suppose this claim does not hold for state s.

$$P_s \not\sqsupseteq_R h\ s \implies \exists p \in h\ s, \forall p' \in P_s, \neg p R p' \tag{18}$$

Suppose partial solution p is constructed from partial solutions $p_1, \ldots, p_k$ where p_i is taken from state s_i. By the inductive hypothesis, for each $i \in [1, k]$, there exists $p'_i \in r\ s_i$ such that $p_i ?R p'_i$. Let $\overline{p} = (p_1, \ldots, p_k)$ and $\overline{p'} = (p'_1, \ldots, p'_k)$.

$$\begin{aligned}
\bigwedge_{i=1}^{k} p_i ?R p'_i \implies &\forall p'_1, \left(\overline{p} \twoheadrightarrow_{h,s} p'_1 \rightarrow \exists p'_2, \overline{p'} \twoheadrightarrow_{h,s} p'_2 \wedge p'_1 ?R p'_2\right) \\
\implies &\exists p'_2, \overline{p'} \twoheadrightarrow_{h,s} p'_2 \wedge p ?R p'_2 \\
\implies &\exists p'_2 \in P_s, p ?R p'_2
\end{aligned} \tag{19}$$

Formula 19 contradicts with Formula 18. Therefore, we prove the second claim, and thus the induction holds. □

Lemma A.3 (Lemma 4.1). *Given instance i, for any two keyword preorders R_1, R_2 where all comparisons in R_1 are included in R_2, the following formula is always satisfied.*

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

where R_2/R_1 represents the keyword preorder formed by the comparisons in R_2 that are not used in R_1.

Proof. We start with the $\leftarrow$ direction. Suppose there is an example e in $CE(R_1, i)$ satisfying $\exists(p_1, p_2) \in e, \neg p_1(R_2/R_1)p_2$. As the comparisons in R_2/R_1 are included in R_2, this premise implies $\exists(p_1, p_2) \in e, \neg p_1 R_2 p_2$. By Formula 11, 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 $(p_{1,1}, p_{2,1}), \ldots, (p_{1,n}, p_{2,n})$ 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. Be the definition of CE, we have (1) $\forall(p_1, p_2) \in e, p_1 R_1 p_2$, (2) the following formula.

$$\begin{aligned} & \exists p_1', \overline{p_1} \twoheadrightarrow_s p_1' \wedge \forall p_2', \left(\overline{p_2} \twoheadrightarrow_s p_2' \rightarrow \neg p_1' R_1 p_2'\right) \\ \Longrightarrow\ & \exists p_1', \overline{p_1} \twoheadrightarrow_s p_1' \wedge \forall p_2', \left(\overline{p_2} \twoheadrightarrow_s p_2' \rightarrow \neg p_1' R_2 p_2'\right) \end{aligned} \tag{20}$$

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

$$\begin{aligned} & \forall(p_1, p_2) \in e, p_1 R_1 p_2 \wedge \forall(p_1, p_2) \in e, p_1(R_2/R_1)p_2 \\ \Longrightarrow\ & \forall(p_1, p_2) \in e, \forall(op, k) \in R_2, (k\ p_1) op (k\ p_2) \\ \Longrightarrow\ & \forall(p_1, p_2) \in e, p_1 R 2 p_2 \end{aligned} \tag{21}$$

Combining Formula 21 with 20, we know example e is in $CE(R_2, i)$, and the other direction of this lemma is proved. □

Lemma A.4 (Lemma 4.2). *Given a set of instances I, for any two keyword preorders R_1, R_2 where all comparisons in R_1 are included in R_2 and $\forall i \in I, CE(R_2, i) = \emptyset$, 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.,*

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

where $|R|$ represents the number of comparisons in keyword preorder R.

Proof. Let $(op_1, k_1), \ldots, (op_n, k_n)$ 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.

$$\exists x \in [1, n], \left|CE(R_1, I) \Big/ CE(R_x^a, I)\right| \geq \frac{|CE(R_1, I)|}{n} \tag{22}$$

We prove Formula 22 in two steps. First, we prove that $\forall x \in [1, n]$ satisfies the following formula.

$$\left|\Big(CE(R_x^p, I)/CE(R_1, I)\Big) \Big/ \Big(CE(R_{x-1}^p, I) \Big/ CE(R_1, I)\Big)\right| \leq \left|CE(R_1, I) \Big/ CE(R_x^a, I)\right| \tag{23}$$

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. Then, by Lemma 4.1,

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

Therefore, $|C_x^p| \leq |C_x^a|$ and thus Formula 23 is proved.

Then, we prove the following formula.

$$\exists x \in [1, n], |C_x^p| \geq \frac{|CE(R_1, I)|}{n} \tag{24}$$

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

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

Therefore, we prove that Formula 24 holds for $x = x^*$.

As the combination of Formula 23 and Formula 24 implies Formula 22, the target lemma is proved. □

Theorem A.5 (Theorem 4.3). *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) $\forall i \in I, CE(R, i) = \emptyset$, and (2) R is constructed by $(\leq, o)$ and some comparisons in G, MetHyl must terminate and return such a keyword preorder.*

Proof. Let R be any solution satisfying the three conditions. According to Algorithm 1, given a finite set of comparisons and a size limit, function `BestPreorder` always terminate.

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

Let $(op_1, k_1), \ldots, (op_n, k_n)$ be an order of comparisons used in $R/\{(\leq, o)\}$ such that for any $x \in [1, n]$, (op_x, k_x) will be a valid comparison for function `CandidateComps` in the xth turn if $(op_1, k_1), \ldots, (op_{x-1}, k_{x-1})$ are selected in the previous terms. According to Lemma 4.1, 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 $\forall x \in [1, n]$, (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. □

Lemma A.6 (Lemma 5.1). *Given instance i and program $prog_1$ in Form 10, let $prog_1'$ be result of Rule 1. If for any query q and constructor m, Formula 13 and Formula 14 are satisfied respectively, $prog_1 \sim_i prog_1'$ holds.*

$$\forall e \in RE(q, i), q\ e = ?q[q]\ (\mathsf{F}[q]?f_p\ e) \tag{25}$$

$$\forall e \in RE(m, i), ?f_p\ (m\ e) = ?c[m]\ (\mathsf{F}[m]?f_p\ e) \tag{26}$$

Proof. Recall the form of $prog_1$ and $prog_1'$ as the following.

$$prog_1 = (rg((thin\ ?R) \circ cup \circ \mathsf{P}\phi, \psi)_{\mathsf{F}}, o)$$
$$prog_1' = (rg((thin\ R') \circ cup \circ \mathsf{P}\phi', \psi)_{\mathsf{F}}, ?q[o])$$

Comparing $prog_1'$ with $prog_1$, 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 $prog_1$ and $prog_1'$ 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 as $e_2[k]$ after (1) replacing all solution-related functions with the corresponding $?q$ and $?c$, and (2) replacing all solutions with the outputs of $?f_p$.

If this claim holds, the last programs in e_1 and e_2 must be the same because they are the outputs of $prog_1$ and $prog_1'$ and include neither functions nor solutions. So this claim implies $prog_1 \sim_i prog_1'$.

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 $prog_1$.

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. Therefore, the examples used in the synthesis task of Step 2 ensures that the outputs are still corresponding. 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. □

Lemma A.7 (Lemma 5.2). *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 $s_1, s_2 \in (S_r\ i)$, $r\ s_1 \neq r\ s_2 \to ?f_m\ s_1 \neq ?f_m\ s_2$.*

Proof. Consider the following claim.

- Each time when $r^{?f_m}\ s$ returns, (1) the results 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 $r^{?f_m}\ s_1, \ldots, r^{?f_m}\ s_n$ be all invocations of $r^{?f_m}$ during $r^{?f_m}\ i$ and suppose they are ordered according to the returning 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 $r^{?f_m}$ returns on s_k.

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 memoized result in $r^{?f_m}\ s_k$.

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

□

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 generated by MetHyl$^+$ with I. 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 Theorem 3.7, Lemma 5.1, Lemma 5.2, and the correctness of Step 4. □

Theorem A.9 (Theorem 5.4). *Given input $([\![\phi, \psi]\!]_F, o)$ and grammar G specifying the program space for synthesis tasks, the program generated by MetHyl$^+$ must be pseudo-polynomial time if the following conditions 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 on the generator, and (4) the time complexity of each invocation on the scorer. To prove this theorem, we only need to prove that all of these four factors are pseudo-polynomial time.

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 each 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 range of $?f_m$, which is equal to the product of the ranges of key functions in $?f_m$. The second factor is bounded by the number of partial solutions returned by $thin[?R]$. By Theorem 3.6, this value is also bounded by the product 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 range of each key function 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 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 range of each key function in $?R$ is pseudo-polynomial.

Third, we prove that the third factor is pseudo-polynomial. According to Section 5, the generator in the resulting program must be in the following form:

$$rg(thin[?R] \circ cup \circ \mathrm{P}\phi', \psi')$$

Therefore, the time complexity of each invocation can be further decomposed into four factors: (3.1) the time cost of $thin[?R]$, (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 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, 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 also pseudo-polynomial because (1) the number of solutions and the scale of solutions are both pseudo-polynomial, and (2) the time complexity of the 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. □