Synthesizing Efficient Dynamic Programming Algorithms

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

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

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

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

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

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

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

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

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

Concretely, our approach takes a (possibly inefficient) recursive program specified as a (relational) hylomorphism and produces an efficient memoization program. A hylomorphism is a program written in a specific yet general recursive pattern. Meijer et al. [1991] show that every primitive recursive function can be written in a hylomorphism. Different approaches for automatically deriving hylomorphisms have been proposed, including generating a hylomorphism from
a recursive program [Bird and De Moor 1994; Hu et al. 1996] or from a logic specification [Lin et al. 2021].

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

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

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

2 Overview

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

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

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

To formally describe this COP, we need to specify (1) the set of valid solutions (in this case, subsets of items whose total weight is within the capacity) and (2) the objective value of each solution (in this case, the total value of each subset).

In MetHyl, the user writes two programs, generator \( g \) and scorer \( o \), to specify the two parts. The generator \( g \) takes the parameters of the problem (in this case, the list of items) as input and generates all valid solutions for the problem. The scorer \( o \) is an objective function that maps each solution to its objective value.

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

\[
\begin{align*}
g &= \lambda g.\lambda x.s.\text{ if } |x| = 0 \text{ then } [] \leftarrow \text{else } (g \,(\text{tails } x)) \leftrightarrow \\
& \quad \left\{ \text{head } x \:: p \mid p \in g \,(\text{tails } x), \right. \\
& \quad \left. (\text{sumw } p) + (\text{head } x), 1 \leq W \right\} \\
o &= \lambda p.(\text{sumv } p)
\end{align*}
\]

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

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

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

\[
\begin{align*}
m &= \lambda x.s.xs \\
mem &= \lambda mem.\lambda x.s.\text{ if } \text{buffer}[m \, x] = \bot \text{ then } \\
& \quad \text{buffer}[m \, x] \leftarrow g \, \text{mem } x; \\
& \quad \text{buffer}[m \, x] \\
prog &= \lambda x.s.\text{argmax } o \,(\text{fix } \text{mem } x) \, x)
\end{align*}
\]

Though the memoization algorithm \( \text{prog} \) has effectively reused the repeated invocations to \( g \), its time complexity is still exponential to the number of items. The goal of our approach is to optimize this inefficient memoization algorithm into an efficient algorithm.
2.1 Transformation System

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

\[
O \left( n_{\text{keys}} \left( s_d + n_{\text{sols}} s_{\text{sols}} \right) \right)
\]

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

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

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

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

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

\[
p_1 R p_2 \iff \text{sumw } p_1 \geq \text{sumw } p_2 \land \text{sumv } p_1 \leq \text{sumv } p_2
\]

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

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

**Step 2.** This step optimizes $s_{\text{sols}}$, the size of a partial solution, using incrementalization proposed by Liu and Stoller [2003].

The goal of incrementalization is to change the representation of a solution from a list to another data structure, so that $s_{\text{sols}}$ could be reduced.

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

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

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

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

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

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

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

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

2.2 Correctness Specification Extraction for Program Synthesis

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

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

1. Let $p \rightarrow_s p'$ denote a partial solution $p'$ for state $s$ is constructed from another solution $p$. Let $(g_1, xs), (g_1, \text{tail } xs)$
be two consecutive invocations, \( p_1, p_2 \in g_1 \) (tail \( xs \)) and \( p'_1 \in (g_1 \) \( xs \)) be three solutions returned by the two invocations such that \( p_1 \rightarrow_{xs} p'_1 \). The following condition holds.

\[
\begin{align*}
p_1 & \rightarrow R p_2 \\
\text{if} & \quad [0,0] \\
\text{else thin } & \quad R (g_1 \) \( tail \) \( xs \)) \\
\text{then } & \quad R (g_1 \) \( tail \) \( xs \)) \\
& \quad [c (head \) \( xs \)) \p \in g_1 \) \( tail \) \( xs \)) \left( p_1 + (head \) \( xs \))1 \leq W \right)
\end{align*}
\]

\[
\begin{align*}
q_1 & = \lambda p. (sumwp) \quad m_1 = \lambda x. xx \\
R & = \lambda p. (sumwp) \quad p_1 \geq p_2 \land \sumwp p_1 \leq \sumwp p_2
\end{align*}
\]

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

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

The first condition ensures that \( g'_1 \) \( xs \)) is always equal to \( (\text{thin } R (g_1 \) \( xs \)) \), and the second condition ensures that the solution with the largest objective value is always reserved. Therefore, they together imply the correctness of \( (g'_1, o'_1, m'_1) \).

To ensure the second condition, MetHyl limits the form of \( R \) to be \( \lambda p_1, p_2.0 \leq o_2 \land p_1 \rightarrow R p_2 \), where \( \rightarrow R \) is another unknown preorder, and then requires only the first condition.

**Step 2 and Step 4.** In these two steps, the synthesized function should behave the same as the original function. However, since the input and output type of the synthesized function may be different from the original version, we rely on the synthesized converting function for the comparison. Suppose \( A \) is the type to be replaced and \( \rightarrow R \) is the converting function to be synthesized. For example, given a function \( q \) taking \( A \) as input, the synthesized function \( \rightarrow R q \) should satisfy

\[
\begin{align*}
\text{synthesized function } & \quad \rightarrow R q = \lambda p. (sumwp) \quad o_2 = \lambda x. xx \\
\text{example } & \quad (q_1 \) \( tail \) \( xs \)) \left( p_1 + (head \) \( xs \))1 \leq W \right)
\end{align*}
\]

Our synthesis algorithms use inductive synthesis [Jha and Seshia 2017], and thus we need to collect examples for synthesis. To make the synthesis effective, the examples should reflect a situation that could possibly appear during the program execution. Therefore, we randomly generate inputs to the whole program, and collect runtime values related to the specification.
To enable efficient synthesis for Step 1 and Step 3, we eliminate the logical implication in their specifications and collect simpler examples for \(?R\) and \(?f_m\).

For Step 1, the fixed form of \(?R\) implies the following property.

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

With the first condition, we obtain the following formula.

\[
(\forall p'_1 \in (g_1 \ x_1), \neg p_2 \rightarrow x_1 \ p'_2 \lor o \ p'_1 > o \ p'_2) \rightarrow \neg p_1 \ ?R \ p_2
\]

The premise of the implication does not contain \(?R\), and thus we can collect all pairs of partial solutions \((p_1, p_2)\) where there exists \(p'_1\) to satisfy the premise. In this way, we obtain a set of negative examples. Note that when other parts of \(?R\) are synthesized, MetHyl will collect a new set of examples similarly. More details can be found in Section 5.1.

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

\[
g_3 \ x_{s1} \neq g_3 \ x_{s2} \rightarrow \ ?f_m \ x_{s1} \neq \ ?f_m \ x_{s2}
\]

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

### 2.3 Synthesizing Fragments from Examples

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

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

\[
a \ R \ b \iff \land ((\text{key}, a) \ ?op_i \ (\text{key}, b))
\]

where \(?op_i\) is in \((\leq, =)\) and \((=)\) in Step 1 and 3 respectively.

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

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

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

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

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

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

### 3 Preliminaries

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

\[
(f_1 \circ f_2) \ x := f_1(f_2 \ x) \quad (f_1 + f_2)(i, x) := f_1 \ x, i \in \{0, 1\}
\]

\[
(f_1 \times f_2)(x, y) := (f_1 \ x, f_2 \ y) \quad (f_1 \triangle f_2) \ x := (f_1 \ x, f_2 \ x)
\]

#### 3.1 Categorical Functors

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

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

\[
F(id_A) = id_{F A} \quad F(f \circ g) = Ff \circ Fg
\]

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 \(\times, \circ\).
Their definitions are shown below.
\[\text{IA} := A \quad \text{if} \ f := \text{id}_A \quad (\text{IA})B := A \quad (\text{IA})f := id_A\]
\[(F_1 \times F_2)A := F_1A \times F_2A \quad (F_1 \times F_2)f := F_1f \times F_2f\]
\[(F_1 + F_2)A := \{1\} \times F_1A \cup \{2\} \times F_2A\]
\[(F_1 + F_2)f := F_1f + F_2f\]
where \(A \times B\) is the Cartesian product of objects \(A\) and \(B\).

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

\[PA := \{s \mid s \subseteq A\} \quad Pf s := \{fa \mid a \in s\}\]

### 3.2 Generator, Memoization, and Hylomorphism

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

**Definition 3.1** (Recursive Generator). Given arrows \(\phi : PFA \rightarrow PA\) and \(\psi : B \rightarrow PFB\), recursive generator \(rg(\phi, \psi)_F : A \rightarrow PB\) is the smallest solution of the following equation, where arrow \(r_1\) is smaller than \(r_2\) if \(\forall i, r_1 i \subseteq r_2 i\).

\[rg(\phi, \psi)_F = \phi \circ \text{cup} \circ \text{car}[F] \circ \text{Frg}(\phi, \psi)_F \circ \psi\]

In this equation, \(\text{car}[F] : PFA \rightarrow PFA\) is defined as the following, which operates similarly with the Cartesian product, and \(\text{cup} : PPA \rightarrow PA\) is defined as \(\text{cup} x := \cup_{a \in x} a\), which returns the union of all sets in a set of sets.

\[\text{car}[I] x := x \quad \text{car}[F_1 \times F_2] (x_1, x_2) := (\text{car}[F_1] x_1) \times (\text{car}[F_2] x_2)\]

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

**Definition 3.2** (Relational Hylomorphism). Given two arrows \(\phi : FA \rightarrow PA\) and \(\psi : B \rightarrow PFB\), relational hylomorphism \([\phi, \psi]_F : B \rightarrow PA\) is defined as \(rg(\text{cup} \circ \text{P} \phi, \psi)\).

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

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

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

### 3.3 Programs

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

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

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

### 3.4 Thinning

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

\[
\forall a \in A, aRa \quad \forall a, b, c \in A, aRb \land bRc \rightarrow aRc
\]

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

One noticeable special case of \(\text{thin}\) is when \(R\) is the conjunction of several comparisons on several key functions.

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

\[aRb \iff \land_i(k_i a \land op_i(k_i b)\]

where \(k_i\) is an arrow from \(A\) to \(\text{Int}\) and \(op_i \in \{\leq, =\}\).

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

\[N_R(S) := \left\lfloor \max_{i} \left(\text{range}(k_i, S) \mid \text{op}_i \in \{\leq, =\}\right) \right\rfloor + \text{max}(S)\]

- For any set \(S\), \(\mid \text{thin } R S \mid \leq N_R(S)\).
- There is an implementation of \((\text{thin } R)\) with time complexity \(O(N_R(S) \text{size}(R) + T_P(R))\), where \(S\) is the input set, \(\text{size}(R)\) is the number of comparisons in \(R\), \(T_P(R)\) is the time complexity of evaluating all key functions in \(R\) for all elements in \(S\).

Due to the space limit, we omit the proofs to theorems, which can be found in Appendix A.
\(prog = ([\{\phi, \psi\}]_F, o), \) where
\[F = \text{Unit} + 1 + ((\text{Int} \times \text{Int}) \times 1)\]
\(\psi = \text{last}, \text{if } |xs| = 0 \text{ then collect } (1, \text{unit});\)
else \{collect (2, tail xs); collect (3, (head xs, tail xs)); \}
\(\phi = \lambda (\text{tag, p}), \text{if } \text{tag} = 1 \text{ then collect } [];\)
else if \(\text{tag} = 2 \text{ then collect } p;\)
else if \((\text{sumw p}) + p.1 \leq W\) then collect \((p.1 :: p.2);\)
o = \lambda p. \text{sumv p}

**Figure 2.** The program corresponding to \((g_1, a_1)\).

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

**Figure 3.** A valid program for 0/1 knapsack in \(\mathcal{L}_H\).

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

**Definition 3.7.** Given arrows \(p, h\) starting from object \(A\), a set \(M\) including \(n\) arrows \(m_i : F_{m_i} A \rightarrow A\), and example space \(E\) attaching a set of examples \(E_{m_i}\) to each \(m_i \in M\), lifting problem \(\text{LP}(M, p, h, E)\) and partial lifting problem \(\text{PLP}(M, p, h, E)\) are to find \(?f, ?c_1, \ldots, ?c_n\) such that Equations 1 and 2 are satisfied for all \(m_i \in M\), respectively.

\[\forall e \in E_{m_i}, ((p \circ ?f) \circ m_i) e = (?c_i \circ F_{m_i}(h \circ ?f)) e \quad (1)\]
\[\forall e \in E_{m_i}, (p \circ m_i) e = (?c_i \circ F_{m_i}(h \circ ?f)) e \quad (2)\]

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

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

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

4.1 Step 1: Thinning
Given program \((h = [[\phi, \psi]]_F, o)\) and a set of instances \(I\), MetHyl returns a program in the following form while keeping the correctness on set \(I\), where \(?R\) is a keyword preorder.

\[(rg((\text{thin } ?R) \circ \cup \circ \phi, \psi)_F, o)\] (3)

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

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

\[\forall s \in (S_i, i), \forall s' \in (T_h s), \forall p_1, p_2 \in (h s'), p_1 ?R p_2 \rightarrow \forall p_1', (p_1 \rightarrow s, p_1' \rightarrow \exists p_2', p_2 \rightarrow s, p_2' ?R p_2')\] (4)
Given a preorder $R$ and an instance $i$, a set of counterexamples $CE(R, i)$ can be extracted via Lemma 4.1, which includes all pairs of partial solutions $(p_1, p_2)$ that violates Equation 4. Note that $CE(R, i)$ changes with the choice of $R$.

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

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

\[ \forall (p_1, p_2) \in CE(R_1, i), (p_1, p_2) \notin CE(R_2, i) \rightarrow \neg p_1(R_2/R_1)p_2 \]

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

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

\[
\text{cost}(?R, I) := N_{CE}(P), \quad P = \{p | i \in I, s \in S_h, p \in h s\}
\]

where $P$ represents all partial solutions related to set $I$.

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

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

### 4.2 Step 2: Incrementalization for Plans

The initial program for Step 2 is in Form 3, where the body of $\phi$ and $\psi$ are statements ($S$) in language $L_H$ (Figure 4).

MetHyl performs Step 2 only when the partial solutions generated by $\phi$ includes recursive data structures such as lists and trees. Concretely, MetHyl constructs a program $prog'_2$ that performs almost the same with $prog_2$ except each partial solution $p$ in $prog_2$ is stored as $?fp$, where the output of $?fp$ is limited to a tuple of scalar values. The construction rewrites two types of solution-related functions in $prog_2$.

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

The following shows the content of $prog'_2$.

\[
prog'_2 = \text{rg}((\text{thin } R') \circ \cup \circ P\phi', \psi, ?q_0) \quad (5)
\]

- Both $o$ and keywords in $?R$ are queries. In $prog'_2$, $o$ and $R$ are replaced with $?q_0$ and $R' = \{ (op, ?q[k]) | (op, k) \in R \}$.
- Constructors and other queries are extracted from $\phi$. In $prog_2$, $\phi$ is replaced with function $\phi'$ constructed from $\phi$. $\phi'$ is constructed via a structural recursion on the AST of $\phi$. This procedure is shown as Algorithm 1.

**Algorithm 1:** Construct $\phi'$ from $\phi$.

| Input: A program $\phi = \lambda s \phi_s$, where $\phi_s$ is a statement in $L_H$. |
| Output: Queries $Q$, Constructors $M$, and $\phi'$ in $prog'_2$. |

1. $Q \leftarrow \emptyset$; $M \leftarrow \emptyset$;
2. Function $\text{RewriteE}(p_e)$:
   - if $|s| \notin p_e$ then return $p_e$;
   - if $p_e = |s|$ then return $\perp$;
   - $sp_e' \leftarrow \{ \text{RewriteE}(sp_e) | sp_e \in \text{Children}(p_e, \emptyset) \}$;
   - if $\perp \in sp_e'$ then
     - if $p_e$ output a tuple of scalar values then
       - $t_1, \ldots, t_m \leftarrow$ temporary variables in $p_e$;
       - $Q.\text{Insert}(p_e)$; return $?q[p_e](s(t_1, \ldots, t_m))$;
     - return $\perp$;
   - end
   - return $\text{Clone}(p_e, sp_e')$;
3. Function $\text{RewriteS}(p_s)$:
   - if $p_2 = \{ \text{collect } sp_e \}$ then
     - $t_1, \ldots, t_n \leftarrow$ temporary variables in $p_s$;
     - $M.\text{Insert}(sp_e)$; return $?[c[p_s](s(t_1, \ldots, t_n))$;
   - end
   - $sp_e' \leftarrow \{ \text{RewriteS}(sp_e) | sp_e \in \text{Children}(p_s, \emptyset) \}$;
   - $sp_e' \leftarrow \{ \text{RewriteS}(sp_e) | sp_e \in \text{Children}(p_s, S) \}$;
   - return $\text{Clone}(p_s, sp_e' + sp_e)$.
4. $\phi' \leftarrow \text{RewriteS}(\phi_s)$; return $Q, M, \lambda s \phi'_s$.

To characterize the specification of $?fp$, $?q$ and $?c$, we introduce two notations $F[p]$ and $E(p, t)$.

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

Lemma 4.3 provides a sufficient condition for $?fp$, $?q$ and $?c$ to guarantee the correctness of $prog'_2$.

**Lemma 4.3.** Given instance $i$ and program $prog_2$ in Form 3, let $prog'_2$ be the program constructed by MetHyl in Step 2. $prog_2 \sim _1 prog'_2$ if for any query $q$ and any constructor $m$, Formula 6 and Formula 7 are satisfied respectively.

\[
\forall e \in RE(q, i), q e = ?q[p] \quad (6)
\]

\[
\forall e \in RE(m, i), ?fp (m e) = ?c[m] \quad (7)
\]

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

- (Correctness) Lemma 4.3 is satisfied for any $i \in I$.
- (Efficiency) The output of $?fp$ is a tuple of scalar values.
4.3 Step 3: Memoization

In Step 3, given program \((r, o)\), where \(r\) is a recursive generator, and a set of instances \(I\), Methyrl returns \((r'^{fm}, o)\), i.e., memoizes \(r\) while keeping the correctness on set \(I\).

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

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

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

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

\[
\text{cost}(?f_m, I) = \max\{|?f_m s| \mid i \in I, s \in (S, i)\}
\]

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

- (Correctness) \(\forall i \in I, \forall (s_1, s_2) \in EM(i), \text{cost}(?f_m s_1) = \text{cost}(?f_m s_2)\).
- (Efficiency) \(\text{cost}(?f_m, I)\) is minimized.

4.4 Step 4: Incrementalization for States

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

\[
\text{prog}_4 = (r, o), \quad r = rq((\text{thin } R) \cup P \phi, \psi)^{fm}_F
\]

where the body of \(\phi\) and \(\psi\) are statements \((\subseteq)\) in \(L_H\).

Methyrl performs Step 4 only when the type of states includes recursive data structures. Similar to Step 2, Methyrl rewrites all state-related queries and constructors in \(\text{prog}_4\) with \(?q\) and \(?c\) respectively, and thus constructs a program \(\text{prog}_4'\) that performs almost the same with \(\text{prog}_4\) except each state \(s\) in \(\text{prog}_4\) is stored as \(?f_s s\). To guarantee the efficiency, the output of \(?f_s\) is limited to a tuple of scalar values.

In \(\text{prog}_4'\), key function \(f_m\) is identified as a query. Other queries and constructors are extracted from \(\psi\) similarly with Algorithm 1. The only difference is that in \(\psi\), the parameter \(s_{pe}\) of \([\text{collect } spc]\) is a structure including substates, on which \(r\) will be recursively applied, and decision values, which will be directly passed to \(\phi\). Because \(F\) is a polynomial function, these components can be extracted from \(spc\) via the access operator, i.e., \([spc t_1 t_2 \ldots t_n]\). Those components corresponding to substates and decision values are identified as constructors and queries respectively.

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

5 Synthesizer

5.1 Preorder Synthesis

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

As shown in Algorithm 2, Methyrl solves this subproblem via branch-and-bound. BestPreorder decides comparisons used in \(?R\) one by one with an upper bound on the cost (Lines 1-14). In each turn, a set of candidate comparisons is identified via CandidateComps (Line 5) and they are considered in the increasing order of the size (Line 6). For each comparison \(c\), if its cost is smaller than the bound (Line 9), preorders including \(c\) will be considered recursively (Line 10). Results of recursions will be used to update the bound (Line 11). According to Lemma 5.1, CandidateComps \((C, lim, es)\) returns comparisons that satisfy at least \(|es|/lim\) examples.

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

\[
\left| \{(p_1, p_2) \in CE(R_1, I) \mid (k \circ p_1) \circ p_2 \} \right| \geq \frac{|CE(R_1, I)|}{|R_2| - |R_1|}
\]

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

Decide a finite set of comparisons. In practice, the set of comparisons is usually provided by a grammar, in which the number of comparisons is infinite. Due to the difficulty of finding the optimal preorder from an infinite set of comparisons, Methyrl approximates it via the principle of Occam’s Razor. Concretely, Methyrl selects a parameter \(s_c\) and takes the smallest \(s_c\) comparisons in the grammar as set \(C\).
Decide $s_c$ and $n_c$. In practice, both $s_c$ and $n_c$ are not given. MetHyl decides them iteratively with two parameters $s'_c$ and $n'_c$. In each turn, MetHyl considers the subproblem where $n_c = n'_c$ and $s_c = s'_c$. If there is no solution, $s'_c$ will be doubled and $n'_c$ will be increased by 1 in the next iteration.

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

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

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

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

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

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

5.2 Synthesis Incrementalization

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

$$(q \circ id) e = (?q[q] \circ F[q](null \Delta ?f_q)) e$$  (8)

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

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

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

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

$$\forall m \in M, (f_q \Delta ?f_q) \circ m) e = (?c[m] \circ F[m](f_q \Delta ?f_q)) e$$  (9)

where $M$ represents the set of constructors.

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

5.3 Properties of MetHyl

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

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

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

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

6 Implementation

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

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

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

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

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

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

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

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

7 Evaluation
Our evaluation answers two research questions.

• \textbf{RQ1:} How is the overall performance of \textit{MetHyl}?

• \textbf{RQ2:} How do the transformation steps in \textit{MetHyl} perform?

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

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

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

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

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

The second category includes COPs where solutions are constrained. A representative COP is \textit{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 ($g = \{[\phi, \psi]_t, o\}$) according to three extreme principles, which correspond to the three programs in Figures 5, 2, 3, respectively.

• The first program keeps all information in the input of \textit{\psi} and filters out invalid solutions while deciding transitions.

• The second program tries all possibilities of constructing solutions in \textit{\psi} and lets \textit{\phi} filter out invalid ones.

• The last program uses \textit{g} to generate all solutions and excludes invalid ones via a small enough objective value.

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

7.3 RQ1: Overall Performance of \textit{MetHyl}
We manually verify the results of \textit{MetHyl} and confirm that all synthesized programs are completely correct. The performance of \textit{MetHyl} is summarized in Table 1. For each problem, COP lists its name as in \textit{Introduction to Algorithm}, \textit{Imp} lists the index of the principle if the problem is in the second category, $T_{\text{inp}}$ lists the time complexity of the input program in $\Omega$ notation, $T_{\text{res}}$ lists the time complexity of the result program in $O$ notation, and \textit{Time} lists the total time used by \textit{MetHyl}.

Table 1 demonstrates both the effectiveness and efficiency of \textit{MetHyl}. On effectiveness, \textit{MetHyl} achieves exponential speed-ups against the input program on 36 out of 37 tasks, accounting for 97.3%.

Besides, we compare the synthesis results with the reference algorithms provided by \textit{Introduction to Algorithm} and a solution to its exercises [Li 2011], and find that \textit{MetHyl} achieves the same complexity on 26 out of 37 tasks, accounting for 70.3%. On efficiency, the average time cost of \textit{MetHyl} to process a program is only 59.2 seconds.

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

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

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

For the other 8 tasks, \textit{MetHyl} fails because of the enumerative solver. Because the scalability of \textit{observational equivalence} is limited, \textit{MetHyl} times out in Step 2 and 4 when the target program is a complex non-linear program, 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 \textit{MetHyl} can be further improved by designing efficient solvers for synthesizing \textit{c}.
Table 1. The overall performance of MetHyl on all tasks in the dataset.

| COP          | Imp | $T_{\text{imp}}$ | $T_{\text{res}}$ | Time(s) | COP          | Imp | $T_{\text{imp}}$ | $T_{\text{res}}$ | Time(s) | COP          | Imp | $T_{\text{imp}}$ | $T_{\text{res}}$ | Time(s) |
|--------------|-----|------------------|------------------|---------|--------------|-----|------------------|------------------|---------|--------------|-----|------------------|------------------|---------|
| 0/1 Knapsack | 1   | $2^n$            | $n^2$            | 25.9    | 1            | $2^n$ | $n^3$           | 133.4            | 15-8    | 1            | $2^n$ | $n^2$           | 31.3             |         |
|              | 2   | 11.9             | 1                | 11.3    | 2            | $n^3$           | 141.4            |         | 2            | $n^2$ | $n^3$           | 25.5             |         |
|              | 3   | 23.8             | 2                | 23.8    | 3            | 240.5            | 12.1              |         | 3            | 4$n^2$ | $n^3$           | 4.8              |         |
| Rod Cutting  | 2$n^2$ | 18.4           | 3                | 24.05   | 15-9         | 4$n^2$ | $n^4$           | 4.8              |         |             |       |                   |                  |         |
| Matrix Chain | $4^n$ | $n^3$           | 15.4             | 40.1    | 15-10        | $n^6$           | 34.1              |         |             |       |                   |                  |         |
| LCS          | 1   | 6.9              | 2                | 41.3    | 1            | $n^6$           | 142.0             |         |             |       |                   |                  |         |
|              | 2   | 5.8$n^2$        | 3                | 274.9   | 15-11        | $n^6$           | 137.5             |         |             |       |                   |                  |         |
|              | 3   | 24.3             | 1                | 41.2    | 3            | $n^6$           | 137.1             |         |             |       |                   |                  |         |
| Optimal BST  | $4^n$ | $n^4$           | 24.5             | 28.1    | 15-12        | $n^6$           | 54.9              |         |             |       |                   |                  |         |
|              | 1   | 8.7              | 2                | 3.6     | 3            | $n^6$           | 27.6              |         |             |       |                   |                  |         |
|              | 2   | 5.6              | 2                | 10.5    | 3            | $n^6$           | 31.6              |         |             |       |                   |                  |         |
|              | 3   | 20.9             | 3                | 8.9     | 4            | $n^6$           |                   |         |             |       |                   |                  |         |

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

Table 2. The performance of each transformation step.

| ID | Exp | Poly | Time | ID | Exp | Poly | Time |
|----|-----|------|------|----|-----|------|------|
| 1  | 31  | 0    | 1.2  | 2  | 0   | 3    | 32.8 |
| 3  | 5   | 0    | 5.1  | 4  | 0   | 24   | 59.2 |

7.4 RQ2: Performance of Transformation Steps

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

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

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

8 Related Work

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

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

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

Program Synthesis. There have been many synthesizers proposed for automatically synthesizing algorithms or efficient programs [Acar et al. 2005; Farzan and Nicolet 2021; Fedyukovich et al. 2017; Hu et al. 2021; Knoth et al. 2019; Morita et al. 2007; Smith and Albarghouthi 2016], but none of them are for dynamic programming.
MetHyl uses program synthesizers to (1) synthesize a key-
word preorder from negative examples, and (2) synthesize
program fragments for incrementalization.

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

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

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

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

9 Conclusion

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

This paper is motivated by the theories of program cal-
culation for deriving efficient dynamic programming algo-
rithms. We notice that there are also studies for deriving
other algorithms such as greedy algorithm [Bird and de Moor
1993; Helman 1989] and branch-and-bound [Fokkinga 1991]
from relational hylomorphisms. Extending MetHyl to sup-
sport 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=ByLdLqvl

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

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

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

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

Oege de Moor. 1994. Categories, Relations and Dynamic Programming.
Math. Struct. Comput. Sci. 4, 1 (1994), 33–69. https://doi.org/10.1017/ S096012950000360

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, 2021, Stephen N. Freund and Eran Yahav
(Eds.). ACM, 974–986. https://doi.org/10.1145/3453483.3454089

Grigory Fedychukov, Maaz Bin Saleer Ahmad, and Rastislav Bodik.
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 Iai 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 Proceedings of the 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759),
Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 783–807.
https://doi.org/10.1007/978-3-030-81685-8_37

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

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

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

A.1 Proofs for Section 3.4

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

$$N_R(S) := \left(\prod_{i=1}^{m} \text{range}(k_i, S)\right) \big/ \max\left(\text{range}(k_i, S) \mid op_i \in \{\leq\}\right)$$

- For any set $S$, $[\text{thin } R S] \leq N_R(S)$.
- There is an implementation of (thin R) with time complexity $O(N_R(S) \text{size}(R) + T_R(S))$, where $S$ is the input set, $\text{size}(R)$ 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., $\arg \max \text{range}(k, S), k \in K_\leq$. 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.

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

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

$$f_k a = f_k b \rightarrow (\text{arR} \leftrightarrow k^* a \leq k^* b)$$

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

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

The remaining task is to prove the correctness of Algorithm 3. Let $A_x$ 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 $\{(op_i, id)_{i=1}^{x}\} \cup \{i=, id\}_{i=x+1}^{m}$. Now, consider the following claim.

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

If this claim holds, after running $A_m$, i.e., Algorithm 3 on $S$, $\text{Val}[f_k a] = a$ 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 the value of $m$. First, when $m$ is equal to 0, this claim holds because only the element with the largest output of $k^*$ is retained while initializing $\text{Val}$ (Lines 6-10).

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

Let $\text{Val}'$ be the value of $\text{Val}$ after running $A_{x-1}$, and let $\text{Val}$ be the value of $\text{Val}$ after running $A_x$. For any $w \in W$ and $i \in [mi_x, ma_x]$, let $w_i$ be the feature that $\forall j \neq i, w_{i,j} = w_{j,i}$. $w_{i,x} = i$. According to Lines 11-20 in Algorithm 3, $\text{Val}[w]$ is equal to the element with the largest output of $k^*$ among $\text{Val}'[w_{w1,i}], \text{Val}'[w_{w1,i+1}], \ldots, \text{Val}'[w_{mx,i}]$. (For simplicity, we define $k^* \perp \bot$ as $-\infty$).

Assume that the claim does not hold for $A_x$. Then, there exists $w \in W$ and $a \in S$ satisfying the following formula.

$$k^* \text{Val}[w] < k^* a \land wR_x(f_k a)$$

$$\Rightarrow k^* \text{Val}'[w_k a] < k^* a \land w_k aR_x-1(f_k a)$$

This fact contradicts with the inductive hypothesis and thus the induction holds, i.e., the claim holds for all $A_x$. \hfill \Box

A.2 Proofs for Section 4.1

In Section 4.1, for simplicity, we assume that each transition, i.e., each element in the output of $\phi$, involves at most one search state. Therefore, in this section, we first extend Section
4.2 to general relational hylomorphisms and then prove the generalized lemmas.

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

Using the generalized notation \(\rightarrow_r\), we generalize Lemma 4.1 to general relational hylomorphisms as the following.

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

\[
\bigwedge_{i=1}^{n} p_{1,i} ? R p_{2,i} \rightarrow \forall p_1', (\overline{p_1} \rightarrow_s s, p_1' \rightarrow \exists p_2', \overline{p_2} \rightarrow_s p_2' \wedge p_1' ? R p_2')
\]

(10)

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

For simplicity, we use \(S_1 \supseteq_R S_2\) 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 \(\text{thin}\), for any preorder \(R\) and any set \(S\), \(\text{thin } R S \supseteq_R S\) always holds.

Let us consider the following claim.

• For any state \(s \in S_h\), \(i, r \subseteq h s \wedge r \subseteq ?R h s\).

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 \(p^*\) is in \(r i\), 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, \(r \subseteq h s\) can be obtained by the definition of \(rg\) and \([\phi, \psi]_F\). Let us unfold the definition of \(h\) and \(r\).

\[
h = \cup \circ \{\phi, \psi\}_F \circ \cup \circ \{\phi, \psi\}_F \circ \{\phi, \psi\}_F \circ \psi
\]

\[
r = \text{thin } R \circ \cup \circ \{\phi, \psi\}_F \circ \cup \circ \{\phi, \psi\}_F \circ \{\phi, \psi\}_F \circ \psi
\]

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

\[
\forall s' \in T_h, r s' \subseteq h s' \\
\Rightarrow \forall t \in \psi, (\text{car } F \circ \text{Fr}) t \subseteq (\text{car } F \circ \text{Fr}) t \\
\Rightarrow (\cup \circ \{\phi, \psi\}_F \circ \text{Fr}) t \subseteq (\cup \circ \{\phi, \psi\}_F \circ \text{Fr}) t
\]

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

The remaining task is to prove \(\forall s \in S_h, r s \supseteq ?R h s\). For any state \(s\), let \(P_i\) be the set of partial solutions constructed in \(r s\) before applying \(\text{thin } R\). Let us consider another claim.

• For any state \(s \in S_h\), \(i, p \supseteq ?R h s\).

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

\[
r s = \text{thin } R P_i \supseteq ?R P_i \supseteq ?R h s
\]

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

\[
P_s \supseteq ?R h s \Rightarrow \exists p \in h s, \forall p' \in P_s, \neg p ? R p'
\]

(11)

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

\[
\bigwedge_{i=1}^{n} p_{1,i} ? R p_{1,i} \rightarrow \forall p_1', (\overline{p} \rightarrow_s s, p_1' \rightarrow \exists p_2', \overline{p'} \rightarrow_s p_2' \wedge p_1' ? R p_2')
\]

(12)

Formula 12 contradicts with Formula 11. Therefore, we prove the second claim, and thus the induction holds. \(\square\)

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

The following is the generalized version of Lemma 4.2.

**Lemma A.3** (Lemma 4.2). Given instance \(i\), for any keyword preorders \(R_1 \subseteq R_2\), the following formula is always satisfied. \(\forall e \in CE(R_1, i), e \notin CE(R_2, i) \Rightarrow \exists (p_1, p_2) \in e, \neg p_1 (R_2/R_1) p_2\)

Where \(R_2/R_1\) represents the keyword preorder formed by the new comparisons in \(R_2\) compared with \(R_1\).

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

For the \(\rightarrow\) direction, suppose there is an example \(e\) in \(CE(R_1, i)\) such that \(\forall (p_1, p_2) \in e, p_1 (R_2/R_1) p_2\).
Let \((p_{1,1}, p_{2,1}), \ldots, (p_{1,n}, p_{2,n})\) be all pairs in example \(e\), let \(p_1\) and \(p_2\) be the sequences of \(p_{1,i}\) and \(p_{2,i}\) respectively. By the definition of \(CE\), we have (1) \(\forall (p_1, p_2) \in e, p_1 R_1 p_2\), and (2) the following formula.

\[\exists p'_1, p_1 \rightarrow s, p'_1 \land \forall p'_2, (p_2 \rightarrow s, p'_2 \rightarrow p'_1 R_1 p'_2)\]

Thus \(\exists p'_1, p_1 \rightarrow s, p'_1 \land \forall p'_2, (p_2 \rightarrow s, p'_2 \rightarrow p'_1 R_1 p'_2)\) (13)

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

\[\forall (p_1, p_2) \in e, p_1 R_1 p_2 \land \forall (p_1, p_2) \in e, p_1 (R_2 / R_1) p_2\]

\[\Rightarrow \forall (p_1, p_2) \in e, \forall (p R, k) \in R_2, (k p_1) R (k p_2)\]

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

\[\square\]

### A.3 Proofs for Section 4.2

**Lemma A.4** (Lemma 4.3). *Given instance \(i\) and program \(prog_2\) in Form 3, let \(prog'_2\) be the program constructed in Step 2, \(prog_2 \sim_i prog'_2\) if for any query \(q\) and any constructor \(m\), Formula 6 and Formula 7 are satisfied respectively.*

\[\forall e \in RE(q, i), q e = \text{?}q[q] (F[q] ?fp e)\]

\[\forall e \in RE(m, i), ?fp (m e) = \text{?}c[m] (F[m] ?fp e)\]

**Proof.** Recall the form of \(prog\) and \(prog'_2\) as the following.

\[prog_2 = (rg((\text{?}R \circ \text{cup} \circ \text{pp} \circ \text{R}, i, o))\]

\[prog'_2 = (rg((\text{?}R' \circ \text{cup} \circ \text{pp}' \circ \text{R}, i, o))\]

Comparing \(prog_2\) with \(prog'_2\), there are several expression-level differences: (1) all key functions in \(?R\) are replaced with the corresponding \(?R'\), (2) the objective function is replaced with \(?q[0]\), (3) all solution-related functions in \(\phi\) are replaced with the corresponding \(?q\) and \(?c\).

Let \(e_1, e_2\) be the small-step executions of \(prog_2\) and \(prog'_2\) on instance \(i\), and let \(e[k]\) be the \(k\)th program in execution \(e\). Let us consider the following claim.

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

If this claim holds, the last programs in \(e_1\) and \(e_2\) must be exactly the same because they are the outputs of \(prog_2\) and \(prog'_2\) and include neither functions nor solutions. Therefore, this claim implies that \(prog_2 \sim_i prog'_2\).

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

Then for any \(k > 0\), consider the \(k\)th 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 \(?fp\) on the partial solutions used in \(e_1\). At this time, the examples used in the synthesis task ensure that the evaluation result is still corresponding.*

*If this evaluation rule does not relate to partial solutions, by the inductive hypothesis, the evaluation in \(e_1\) and \(e_2\) must be exactly the same.*

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

\[\square\]

### A.4 Proofs for Section 4.3

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

**Proof.** Consider the following claim.

*Each time when \(?fm s\) returns, (1) the result is equal to \(r s\), and (2) for any state \(s' \in S_r\), there is result recorded with keyword \(?fm s'\) implies that the results is \(r s'\).*

If the claim holds, the lemma is obtained by \(r^i i = i\).

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

Now, consider the \(k\)th invocation \(r^i s_k\). 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 \(r^i s_k\) returns.

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

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

\[\square\]

### A.5 Proofs for Section 5.1

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

**Lemma A.6** (Lemma 5.1). *Given a set of instances \(I\), for any keyword preorders \(R_1 \subseteq R_2\) where \(\forall i \in I, CE(R_2, i) = \emptyset\), there exists a comparison \((op, k) \in R_2 / R_1\) satisfying at least \(1 / (|R_2| - |R_1|)\) portion of examples in \(CE(R_1, i) = \bigcup_{i \in I} CE(R_1, i)\),*
Therefore, we prove that Formula 19 holds for $\Gamma$.

**Proof.** Let $(o_1, k_1), \ldots, (o_n, k_n)$ be an order of comparisons in $R_1/\{\leq\}$. Let $p_1, p_2, \ldots, p_n$ be the solutions. Then, $p_1(p_2(p_3(\ldots(p_n))))$ will be a valid comparison for CandidateComps in the xth turn if $(o_1, k_1), \ldots, (o_{x-1}, k_{x-1})$ are selected in the previous terms. According to Lemma A.6, such an order must exist.

Suppose the error rate of CandidateComps is at most $c$, i.e., the probability for CandidateComps to exclude a valid comparison is at most $c$. For a good invocation of BestPreorder, $R$ will be found if for any $x \in [1, n]$, $(o_x, k_x)$ is not falsely excluded in the $x$th 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 $t \rightarrow +\infty$, this probability converges to 0.

**A.6 Proofs for Section 5.3**

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

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

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

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

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

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

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 \( c_s f_p(n, w)^s_n \times nw \). In this way, we know that after executing all \( s_p \) operators, the sum of all available values is at most \( c_s f_p(n, w)^s_n \times nw \).

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

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

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

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

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

- According to Theorem 3.6, Factor 3.1 is bounded by the sizes of the ranges of the key functions in \(?R\), which has been proven to be pseudo-polynomial.
- For Factor 3.1 (3.2), the time cost of \( \phi' \) (\( \psi' \)) is bounded by the time cost of \( \phi \) (\( \psi \)) and all inserted program fragments \(?q\) and \(?c\) in Step 2 (Step 4). By the first precondition, their time costs are all pseudo-polynomial with respect to the new state (the new partial solution), which has also been proven to be pseudo-polynomial in both values and scale. Therefore, the time cost of \( \phi' \) (\( \psi' \)) is pseudo-polynomial.
- For Factor 3.3, by the first condition, the number of transitions (denoted as \( n_\ell \)) 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 \( \phi' \), which is bounded by \( n_\ell \times n_p^n \), is also pseudo-polynomial.

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

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

In summary, all four factors are pseudo-polynomial, and thus we prove the target theorem. \( \square \)