Synthesizing Efficient Memoization Algorithms

YICAN SUN, Peking University, China
XUANYU PENG, Peking University, China
YINGFEI XIONG*, Peking University, China

In this paper, we propose an automated approach to finding correct and efficient memoization algorithms from a given declarative specification. This problem has two major challenges: (i) a memoization algorithm is too large to be handled by conventional program synthesizers; (ii) we need to guarantee the efficiency of the memoization algorithm. To address this challenge, we structure the synthesis of memoization algorithms by introducing the local objective function and the memoization partition function and reduce the synthesis task to two smaller independent program synthesis tasks. Moreover, the number of distinct outputs of the function synthesized in the second synthesis task also decides the efficiency of the synthesized memoization algorithm, and we only need to minimize the number of different output values of the synthesized function. However, the generated synthesis task is still too complex for existing synthesizers. Thus, we propose a novel synthesis algorithm that combines the deductive and inductive methods to solve these tasks. To evaluate our algorithm, we collect 42 real-world benchmarks from Leetcode, the National Olympiad in Informatics in Provinces-Junior (a national-wide algorithmic programming contest in China), and previous approaches. Our approach successfully synhesizes 39/42 problems in a reasonable time, outperforming the baselines.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Memoization Algorithms

ACM Reference Format:
Yican Sun, Xuanyu Peng, and Yingfei Xiong. 2023. Synthesizing Efficient Memoization Algorithms. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 225 (October 2023), 35 pages. https://doi.org/10.1145/3622800

1 INTRODUCTION

Combinatorial Problems (CPs). CPs are an essential category of problems that concerns a discrete set of solutions [Schrijver 2003], such as the 01 Knapsack problem [Kellerer et al. 2004], or the longest common subsequence problem [Maier 1978]. It has important applications in various domains. In general, CPs have three major types of problems:

- (Decision) Finding any valid solution.
- (Optimization) Searching for the best valid solution.
- (Counting) Counting the number of valid solutions.

*Corresponding author

Authors' addresses: Yican Sun, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, sycpku@pku.edu.cn; Xuanyu Peng, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Electronic Engineering and Computer Science, Peking University, Beijing, China, dofypxy@pku.edu.cn; Yingfei Xiong, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn.

2475-1421/2023/10-ART225
https://doi.org/10.1145/3622800

Difficulty of Solving CPs. These problems are generally difficult to solve because the number of solutions is tremendous. As a result, we cannot simply enumerate every solution and must design specialized algorithms to solve CPs efficiently.

Dynamic programming (DP) and Memoization. DP is a powerful and widely-used algorithmic approach to solving CPs efficiently [Aho and Hopcroft 1974; Cohen 1983; Cormen et al. 2009]. To design a DP algorithm, we need first to discover hidden structures in the given CP, which requires human insights. Once we have discovered the hidden structures, we could implement DP in a top-down or bottom-up style, and the top-down approach is also termed *memoization.* Intuitively, it stores the results of previous invocations and reuses them when some cached invocation is called again. Typically, it exploits the structure of the costly enumeration algorithm and could be obtained by modifying the enumeration procedure.

Our result. Motivated by the importance of CPs and dynamic programming, we propose *an automated approach to synthesizing efficient memoization algorithms* in this paper. We implement our approach into the tool SynMem. SynMem requires the user to provide a high-level declarative specification of CPs. The user needs to encode the solution as variables and the validity condition of the solution as logic constraints. For optimization problems, the user needs to provide an additional objective function over variables. Such a specification is widely considered as natural for specifying CPs [Adelsberger 2003; Barták 1999]. The output of SynMem is a program representing the synthesized memoization algorithm, which is ensured to be in pseudo-polynomial time.

Existing approaches. There are existing approaches for synthesizing DP algorithms. However, as we know, all existing approaches target a specialized subclass of dynamic programming algorithms and cannot solve many classic problems. For example, Pu et al. [2011]'s approach only supports DP algorithms that use a fixed number of scalar variables for memoization and thus cannot solve the classic 0-1 knapsack problem [Kellerer et al. 2004]; Lin et al. [2021]'s approach does not allow constraints over individual components of the solution, and thus cannot solve the classic longest increasing subsequence problem [Schensted 1961]. More details can be found in Section 7.

Challenges. Hence, it is worth considering automatically synthesizing a memoization algorithm from declarative specifications that applies to various combinatorial problems. However, there are two major challenges.

- *Scalability.* A memoization algorithm is large and is beyond the scalability of existing general program synthesizers.
- *Efficiency.* Most existing program synthesis approaches are designed for functional correctness and do not guarantee the efficiency of the generated program.

Key insights. We take two novel steps to address these challenges.

Step 1. First, we *structure the synthesis* of memoization algorithms by introducing two concepts, namely the local objective function and the memoization partition function. Our structuring incorporates a significantly broader class of memoization algorithms than previous works [Lin et al. 2021; Pu et al. 2011] and captures two fundamental properties of dynamic programming: the optimal substructures and the overlapping subproblems [Cormen et al. 2009].

By the structuring and a sequence of reductions, we reduce synthesizing the memoization algorithms to two *independent* tasks on inductive quantified relational program synthesis [Wang et al. 2018], namely, synthesizing a local objective function (LOF) and synthesizing a memoization partition function (MPF). Each task synthesizes a smaller program fragment. After solving both tasks, we obtain a complete memoization algorithm whose efficiency is controlled by the range of the MPF. The advantages of this step are two-fold.

- First, both tasks produced by this step target a smaller program fragment. Furthermore, the independence between the tasks enables us to solve them separately, yielding an efficient synthesis procedure. This addresses the scalability issue.
- Second, it is hard to synthesize a large program with efficiency guarantees. We reduce controlling the efficiency of the whole memoization algorithm to minimize the range of the function synthesized in the second task. This address the efficiency issue.

Step 2. The specifications of the two synthesis tasks are too complex to be handled by the state-of-the-art solver on the relational program synthesis [Wang et al. 2018]. Thus, we propose a new synthesis algorithm combining inductive and deductive methods. We also use a heuristic method to optimize the range of the program synthesized for the second task. The key insight of this step is two-fold.

- First, the inductive methods help simplify the specification only to contain basic operators, enabling a simple but effective deductive system.
- Second, we apply the deductive methods to bypass the synthesis of a large proportion of unknown functions, reducing the original synthesis task a simpler task that conventional program synthesizers can solve.

Evaluation. To evaluate SynMem, we create a benchmark of 42 declarative specifications of CPs. In detail, we consider the top 36 dynamic programming problems in Leetcode [lee [n. d.]] that appear most frequently in a real interview, all four dynamic programming problems in National Olympiad in Informatics in Provinces-Junior (a national-wide programming contest in China) [NOI [n. d.]] in the past ten years, and all benchmarks in the previous approach [Pu et al. 2011]. SynMem successfully solves 39/42(92.8%) benchmarks in a reasonable time, outperforming the baselines [Pu et al. 2011; Solar-Lezama et al. 2006].

Contributions. To summarize, our work has the following contributions.

- We structure the synthesis of memoization algorithms by capturing their essences. Our structuring is more general than previous works [Lin et al. 2021; Pu et al. 2011]. By a sequence of reductions, we obtain two independent synthesis tasks, which enable us to solve them separately. Furthermore, we can synthesize an efficient MA by minimizing the range of the function to be synthesized in the second task.
- We propose dedicated algorithms to solve two synthesis tasks effectively. The key novelty of our algorithm is the mixture of inductive and deductive methods to take advantage of both types of approaches .
- We create a benchmark of 42 declarative specifications of CPs and evaluate SynMem. The evaluation results show that SynMem could effectively find efficient memoization algorithms compared to baseline synthesis approaches [Pu et al. 2011; Solar-Lezama et al. 2006].

Due to the space limits, we relegate the full version to the author's webpage [ful [n. d.]].

2 OVERVIEW

This part illustrates SynMem via running examples. We provide formal treatments in Sections 4 and 5. Below we first illustrate the specification and the synthesis goal in Section 2.1. Then, we overview SynMem using the classic 0-1 knapsack problem in Sections 2.2–2.6.

2.1 The Specification and The Synthesis Goal

Specification. SynMem accepts purely symbolic declarative specifications of CPs as input, usually written in a constraint modeling language such as MiniZinc [Nethercote et al. 2007]. In these modeling languages, the user only needs to specify the intention of the problem instead of the

```
// Inputs
int: n, C;
array[1..n] of int: weight;
array[1..n] of int: value;
// Solutions
array[1..n] of var 0..1: p;
// Constraints
constraint ∑_{j=1}^{n} weight[j] · p[j] ≤ C;
// Objective
solve maximize ∑_{j=1}^{n} value[j] · p[j];
```

Fig. 1. The specification of KP

detailed algorithm for solving the problem. A typical CP specification consists of four parts. Below, we introduce the four parts with a classic optimization problem, the 0-1 Knapsack Problem (KP), as shown in Figure 1[*]. An instance of KP consists of a knapsack with capacity C and a set of items. Each item has its weight and value. The goal is to choose a subset of items with the maximum total value, such that the total weight of selected items is no more than the capacity C.

1. Inputs. The first part specifies the input parameters of the problem. Assigning different values to the parameters produces different instances of the problem. For KP, we use `n` for the number of items and two arrays, `weight` and `value`, for the weight and the value of the items, respectively. Finally, we use `C` for the knapsack capacity, i.e., the total weight of the items cannot exceed `C`.

2. Solutions. The second part specifies the solution space. In SynMem, a solution consists of several arrays of atomic variables, and the lengths of the arrays and the domain of each atomic variable are finite and depend on the input parameters. In addition, the domain of each atomic variable is a bounded interval of integers. For simplicity, we shall first consider a special case where only one array exists, and discuss how to handle multiple heterogeneous arrays later. In KP, the solution includes a single array `p` where each element `p[i]` is an atomic variable over `0..1`, i.e., a Boolean variable. Variable `p[i]` represents whether to put the i-th item into the knapsack.

3. Constraints. The third part specifies the constraints over the solution space, defining the validity of a solution. In KP, we require the total weight of the chosen items to be no larger than `C`.

4. Objective. For an optimization problem, the fourth part starts with **`solve maximize`** and specifies an objective function that assigns an objective value to each solution. In KP, the objective function returns the total value of chosen items.

Given the four parts, the goal of the optimization problem is to find the maximum objective value that any solution might have, and returns the value. A decision problem or a counting problem is specified by supplying statement **`solve satisfy;`** or **`count satisfy;`** in the fourth part, where the goal is to find any valid solution or to count the number of valid solutions.

Synthesis goal. SynMem aims to synthesize a correct and efficient memoization algorithm (MA) for the given CP. For an optimal problem, the synthesized MA takes the concrete parameters of the problem as input, produces the optimal value of the objective function for this concrete problem instance as output. The algorithm should efficiently obtain the correct result *for every concrete instance*. The synthesis result for KP is presented in Figure 6.

[*]We do not strictly follow the syntax of MiniZinc for conciseness. In MiniZinc, $\sum_{j=1}^{n}$ is written as `sum(j in 1..n)`.

2.2 Structuring MAs

This part gives our structuring of MAs. An MA caches and reuses the results of solved subproblems. To design MAs, we need first to define the notion of subproblems.

Subproblems. Let us first consider using exhaustive search to solve this problem, as shown in Figure 2. The search algorithm recursively enumerates the values of $\texttt{p[1]}, \dots, \texttt{p[n]}$. By enumerating every valid solution, it obtains the maximum objective value. In the program, we use a structure `atom_var_info` to store the meta information of the atomic variables, and `p_info[i].dom` stores the domain of variable `p[i]`, which is `0..1` in this case.

The main procedure `Search` recursively solves a subproblem of the original combinatorial problem, and the definition of a subproblem is shown in Figure 3. A subproblem is a more generalized version of the original combinatorial problem. Compared with the original problem, a subproblem has an additional parameter `i` which separates the original solution array `p[1:n]` into two parts: (i) the prefix $\texttt{p}^\bullet\texttt{[1:i-1]}$ for the enumerated atomic variables, which is in the input part; (ii) the suffix $\texttt{p}^\circ\texttt{[i:n]}$ for the unknown variables to be explored, which is in the solution part. Note that when $\texttt{i} = 1$, the subproblem coincides with the original combinatorial problem.

Figure 4 shows how the subproblems are recursively solved by the exhaustive search using an example. Each non-leaf subproblem generates two child subproblems by considering different assignments to `p[i]`, and their maximum objective value is returned. Each leaf problem corresponds to one solution, and returns either the objective value if the solution is valid or returns $-\infty$.

```
int n,C; int[] weight, value;
atom_var_info[] p_info;
int search(int i, int[] p•) {
    if(i > n)
      return $\sum_{j=1}^{n}$ weight[j]·p•[j] ≤ C ?
             $\sum_{j=1}^{n}$ value[j]·p•[j]: -∞;
    return      $\max_{x \in \texttt{p\_info[i].dom}}$    search(i+1, p•++[x])
}
int main() {
    read(&n, &C, &weight, &value);
    return search(1, []);
}
```

Fig. 2. The search algorithm

```
// Inputs
int: n, C, i;
array[1..n] of int: weight;
array[1..n] of int: value;
array[1..i-1] of 0..1: p•;
// Solutions
array[i..n] of var 0..1: p°;
// Constraints
constraint $\sum_{j=1}^{i-1}$ weight[j]·p•[j]
          +$\sum_{j=i}^{n}$ weight[j]·p°[j] ≤ C;
// Objective
solve maximize $\sum_{j=1}^{i-1}$ value[j]·p•[j]
              +$\sum_{j=i}^{n}$ value[j]·p°[j]
```

Fig. 3. The specification of a subproblem

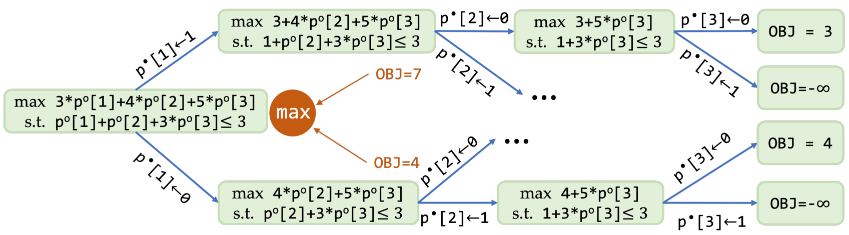

Fig. 4. The subproblems on the parameter $I_0 = (\texttt{n} = 3, \texttt{C} = 3, \texttt{weight} = [1, 1, 3], \texttt{value} = [3, 4, 5])$.

Gaps for memoization. The efficiency bottleneck of the exhaustive search is that the number of subproblems is 2^n. An MA accelerates this process by caching the results of all solved subproblems,

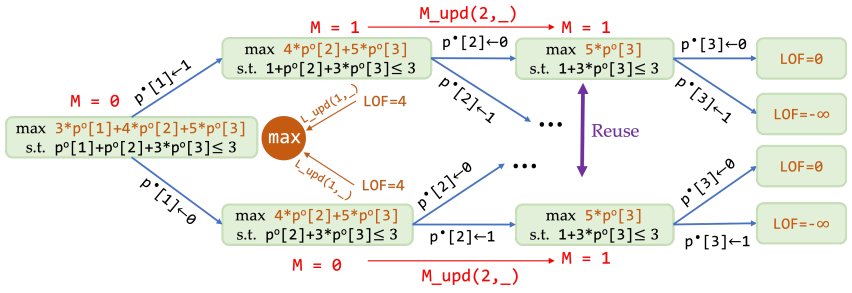

Fig. 5. The execution of the MA on the parameter I_0, where LOF is the LOF, and W is the MPF value.

and reusing a cached result if an equivalent subproblem is encountered. Typically, two subproblems are considered *equivalent* if they have:

- (M1) the same suffix of unknown variables,
- (M2) equivalent objective functions, and
- (M3) equivalent constraints.

where two objective functions are considered equivalent if they map the same solution to the same objective value, and two constraints are considered equivalent if they define the same set of valid solutions.

To form an efficient MA, we need to ensure (1) many pairs of equivalent subproblems exist, (2) the equivalence between subproblems can be efficiently determined without solving the subproblem. Now we analyze the three conditions above. Firstly, consider the condition (M1). Note that the number of subproblems is 2^n, but the number of suffixes is n. Thus, many pairs must have the same suffix. Also, the condition (M1) is easy to check: any pair of subproblems with the same i fulfill this condition. However, the remaining two conditions reflect two gaps in reusing the subproblems in the exhaustive search.

(G1) The objective function for each subproblem depends on the enumerated variables. Since different subproblems have different assignments to the enumerated variables, the objective functions of the two subproblems are unlikely to be equivalent.

(G2) Though many pairs of subproblems satisfy the third condition (M3), since any two subproblems with the same remaining knapsack capability have equivalent constraints, and it is usually assumed that the knapsack capacity C is much smaller than 2^n, it is costly to check whether the constraints of two subproblems are equivalent or not.

To alleviate these gaps, our approach introduces two novel concepts, namely *local objective function (LOF)* and *memoization partition function (MPF)*. The local objective function replaces the original objective function in the subproblems, and does not depend on the enumerated variable. The memoization partition function allows us to easily check the equivalence of constraints between subproblems. Below we describe the two functions in details.

Local Objective Function (LOF). To overcome (G1), we need to make the objective function of the subproblems independent of enumerated variables. SynMem introduces the LOF, which does not depend on the enumerated variables, and replaces the original objective function with the LOF. For any subproblem, the ranking of each solution should not change by changing the objective function to LOF. In addition, the LOF should return the same value as the original objective function on the subproblem where $i = 1$ (i.e., the original problem).

Since LOF does not depend on enumerated variables, any subproblems with the same i have equivalent LOFs. Thus, by replacing the objective function with the LOF, the condition (M2) for equivalent subproblems can be implied by (M1), enabling a higher chance to reuse and an efficient way to check equivalence.

For KP, a LOF can be the second half of the objective function in Figure 3, as follows.

$$\sum_{j=i}^{n} \texttt{p}^{\circ}\texttt{[j]} \cdot \texttt{value[j]}$$

Since the search program recursively solves a subproblem, we need to derive two components from LOF to adapt the search program to return the local objective values.

- An initial local objective value $\texttt{L}_{\texttt{leaf}}$ for the leaf tasks whose corresponding solution is valid. For KP, the initial value is 0, as there is no unknown variable.
- An update function $\texttt{L}_{\texttt{upd}}$ for calculating the local objective value of a solution in a parent subproblem from that of a solution in a child subproblem. For KP, $\texttt{L}_{\texttt{upd}}$ can be $\texttt{L} + \texttt{p}^{\bullet}\texttt{[i]} \cdot \texttt{value[i]}$, where `i` is the input parameter to the parent subproblem and `L` is the optimal LOF value of the child subproblem.

Memoization Partition Function (MPF). To overcome (G2), we need to quickly check the equivalence of the constraints between two subproblems (the condition (M3) above). SynMem introduces the MPF, which is a function over the set of enumerated variables and the input of the original specification. If MPF maps two subproblems with the same parameter `i` to the same value, they share equivalent constraints, which enables efficient checking. SynMem considers MPFs whose output is a tuple of scalars, which is common for MAs. Considering more versatile MPFs is left for future work. For KP, an MPF can be the total weight of currently selected items, as follows.

$$\sum_{j=1}^{i-1} \texttt{weight[j]} \cdot \texttt{p}^{\bullet}\texttt{[j]}$$

However, calculating the MPF is still costly, as we need to scan over all enumerated variables. To further optimize the algorithm, we calculate the MPF incrementally along a sequence of recursions. Similar to the case of LOF, we derive two component from MPF.

- An initial MPF value $\texttt{M}_{\texttt{init}}$ for the first subproblem where $i = 1$. For KP, the initial value is 0, as there is no known variable.
- An update function $\texttt{M}_{\texttt{upd}}$ for calculating the MPF value of a child subproblem from the MPF value of a parent subproblem. For KP, $\texttt{M}_{\texttt{upd}}$ is defined as $\texttt{M} + \texttt{p}^{\bullet}\texttt{[i]} \cdot \texttt{weight[i]}$, where `i` is the input parameter to the child subproblem, and M is the MPF value of a parent subproblem.

The Memoization Algorithm. Based on the LOF and MPF, we can implement an MA for KP. Figure 6 shows a program synthesized by SynMem. For presentation purposes, the program is given as the synthesis result of a sketch. This program follows the same recursive search process as the exhaustive search and introduces a map `mem` to store the result of existing subproblems (Line 3). For non-leaf subproblems, the algorithm first queries `mem` and returns if a cached result of an equivalent problem is found (Lines 12-13). Map `mem` takes the input parameter `i` and the MPF value `M` as the key, which corresponds to the conditions (M1)–(M2) (Note that by the LOF, (M2) is implied by (M1)) and (M3), respectively. The search procedure returns the maximum local objective value rather than the original objective value and updates the LOF and the MPF values incrementally using $\texttt{M}_{\texttt{upd}}$ and $\texttt{L}_{\texttt{upd}}$.

Figure 5 shows how the MA executes on the same example as Figure 4. With LOF and MPF, many subproblems can be identified as equivalent. Note that in each equivalence class of subproblems, we only need to search for the result of one subproblem. We can reuse this result for other subproblems in this class. For MAs, the number of equivalence classes (and thus the number of subproblems to be searched for) is bounded by the domain of the `mem` map, which is a pseudo-polynomial

```
int n, C; int[] weight, value;
atomic_var_info[] p_info;
map<⟨int,int^k⟩, int> mem;
int L_leaf = ○_Lleaf, M_init = ○_Minit;
int L_upd(int i,int L,int x){ return ○_Lupd;};
int M_upd(int i,int^k M,int x){ return ○_Mupd;};
int Search(int i,int^k M,int[] p•) {
    if (i > n)
        if(∑_{j=1}^{n} weight[j]·p•[j] ≤ C)
            return L_leaf;
        else return -∞;
    if(mem.contains(⟨i,M⟩))
        return mem[⟨i,M⟩];
    int ans=       max        L_upd(i,L(x),x)
            x∈p_info[i].dom
     where L(x)=Search(i+1,M'(x),p•_2(x))
     and M'(x)=M_upd(i,M,x)
     and p•_2(x)=p• ++[x]
    return mem[⟨i,M⟩]=ans;
}
```

```
int main() {
    read(&n,&C,&weight,&value);
    initialize(&p);
    return search(1,M_init,[]);
}
```

```
k=1
○_Lleaf=0; ○_Minit=⟨0⟩;
○_Lupd=L + x · value[i];
○_Mupd=⟨M.1+x · weight[i]⟩;
```

Fig. 6. The generated search sketch and the synthesis result for KP. The box in the upper right corner lists the synthesized expressions for $\bigcirc_{\mathtt{Lleaf}}$,$\bigcirc_{\mathtt{Minit}}$,$\bigcirc_{\mathtt{Lupd}}$,$\bigcirc_{\mathtt{Mupd}}$. We can plug these expressions into the sketch to obtain the complete synthesized MA for KP.

$O(\mathtt{n} \cdot \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{weight[i]})$. Furthermore, $\mathsf{L}_{\mathsf{upd}}$ and $\mathsf{M}_{\mathsf{upd}}$ are executed in $O(1)$ time, ensuring a single execution of Search is also efficient. Thus, the MA has the pseudo-polynomial complexity and is significantly more efficient than the exhaustive search with the exponential time complexity.

In the end, we remark the connection between our structuring and two fundamental properties required for dynamic programming.

- *Optimal substructures* requires that a subpart of an optimal solution is still optimal for the subproblem concerning this subpart. In other words, a subproblem should be independent of the enumerated variables. The LOF and MPF jointly ensure this property. LOF removes this dependency in the objective function. MPF ensures that two subproblems with the same parameter i and the same MPF value have equivalent constraints. Thus, the enumerated variables are not needed in checking the validity of a solution when the MPF value is available.
- *Overlapping subproblems* requires that many equivalent subproblems exist. The more subproblems are overlapped, the smaller the number of equivalent classes of subproblems is, and the higher the efficiency of the dynamic programming algorithm is. As discussed above, this number is fully determined by the number of atomic variables and the range of the MPF.

2.3 Overview of The Synthesis Procedure

The synthesis of MAs is non-trivial. As mentioned in the introduction (Section 1), there are two major challenges, scalability and efficiency. An MA is usually a large program with complex loops or recursive calls and is beyond the reach of existing general-purpose program synthesizers [Ji et al. 2021; Lee and Cho 2023; Miltner et al. 2022]. Furthermore, standard synthesis framework such as SyGuS does not require efficiency, and it is not easy to ensure the efficiency of the synthesized MAs.

To address the scalability challenge, we follow our structuring as follows. Synthesizing an MA can be seen as filling a search sketch (Figure 6) that follows an exhaustive search. By generating the search sketch, the scale of the remaining components (the box in the upper right corner in Figure 6) is much smaller. Thus, SynMem first generates the search sketch. Furthermore, SynMem generates the exhaustive search program as the reference implementation to verify the correctness of the synthesized MA. (Section 2.4)

To complete the search sketch, we note that the existing sketch solver [Solar-Lezama et al. 2006] does not scale to synthesize MAs (Section 6). Thus, we further follow our structuring as follows. The key to designing an MA is to find two functions, LOF and MPF, which capture different and independent aspects of the MA. As a result, SynMem converts the sketch synthesis problem into two *independent* problems of synthesizing LOF and MPF. Furthermore, the smaller the range of the MPF is, the more efficient an MA is. Thus, SynMem synthesizes an efficient MA by minimizing the range. After synthesizing the LOF and the MPF, SynMem obtains the corresponding initial values and the updating functions to fill the sketch. (Section 2.5)

Finally, the specifications for the two synthesis problems are *relational* and are too complex to be solved by existing approaches. Thus, we further introduce a new synthesis algorithm that combines inductive and deductive methods. The inductive method removes complex higher-order operators, facilitating deductive analysis. The deductive method applies the term rewriting to bypass the synthesis of a large proportion of unknown functions. We also introduce a heuristic to control the range of the synthesized MPF to ensure efficiency. (Section 2.6).

2.4 Generating the Search Sketch and the Exhaustive Search Program

Generating Process. We generate the exhaustive search and its sketch following a *template-based* method, where most of the code is pre-written and only a few components are generated to adapt to different combinatorial problems. We have seen the search sketch for KP in Figure 6. We can see that most of the search sketch is fixed and can be pre-written, and we only need to generate a few components, all of which can be easily deduced from the specification. Specifically, we need to (i) deduce the size of the solution space and generate the conditional expression in line 8, (ii) generate the conditional expression in line 9 by copying the constraints, (iii) generate line 1 by copying the inputs, (iv) generate the procedure for reading the inputs (line 21), and (v) generate the procedure for initializing `p_info`. The exhaustive search program in Figure 2 can be generated similarly.

Multiple Arrays and Enumeration Orders. The search sketch above only covers the case where there is a single array in the solution part. Below, we illustrate the ideas for extending the search sketch into the case where the solution part contains multiple arrays. In this case, we need to enumerate the atomic variables in all arrays. Therefore, we can treat the solution space generically as a sequence of atomic variables, which is formed by concatenating all the arrays, and the domain of each variable is recorded in `p_info`.

Since the order of enumerating different arrays may affect the synthesis result, SynMem considers the permutations of all arrays, and generates a search sketch for each permutation. Furthermore, sometimes the specification contains multiple arrays of the same length, which are expected to be enumerated together. SynMem also tries zipping the arrays of the same length and enumerates all atomic variables in the same tuple of the zipped array simultaneously.

2.5 Decomposing into Smaller Synthesis Problems

After generating the exhaustive search algorithm and the sketch, we aim to complete the search sketch so that the complete program is efficient and equivalent to the exhaustive search. Such a *synthesis from reference implementation* problem [Farzan et al. 2022; Lee and Cho 2023; Miltner et al. 2022] is often addressed by the counter-example guided inductive synthesis (CEGIS) framework.

CEGIS. CEGIS is performed in iterations. In each iteration, we synthesize an MA that obtains the same output as the exhaustive search on a finite set of problem instances. Then, a verifier verifies whether the synthesized MA is correct. If not, the verifier returns a counter-example, we add the counter-example to the set of problem instances, and start a new iteration.

Our approach is general and can be used with any verifier. Our current implementation uses bounded testing, and exploring more sophisticated verification techniques [Badihi et al. 2020; Churchill et al. 2019] is an orthogonal problem left for future work.

In each CEGIS iteration, the task is to complete the search sketch over a set of instances. As discussed before, this problem is too complex for existing sketch synthesizers, and we first decompose it into independent tasks of synthesizing LOF and MPF.

Specification for the LOF. Though only the initial values and the updating functions are needed in filling the sketch, we synthesize LOF first and then the two components from LOF to reduce the difficulty of synthesis. Recall that the function $\mathtt{LOF(i, p[i:n])}$ needs to satisfy two conditions: 1) returning the same value as the original objective function on the first subproblem, and 2) retaining the order of the solutions. Thus, we impose the following condition (C1) for every instance $\mathtt{(n, C, weight, value)}$ considered in the current CEGIS iteration, and every $1 \le \mathtt{i} \le \mathtt{n}$. Intuitively, this condition requires that the original objective value can be obtained from the local objective value and the enumerated variables.

$$\text{(C1)}\ \exists \oplus_i\ \forall \mathtt{p}.\left(\textstyle\sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{p[j]} \cdot \mathtt{value[j]} = \mathtt{p[1:i-1]} \oplus_{\mathtt{i}} \mathtt{LOF(i, p[i:n])}\right)$$

We further require that $\oplus_i$ is monotonically increasing with respect to the second parameter, i.e., $xs \oplus_i x$ increases as x increases, and $\oplus_1$ should directly return its second parameter, i.e., $[] \oplus_1 x = x$. Note that (C1) is equivalent to the two conditions of LOF, and we are *not* intended to synthesize $\oplus_i$, which is a ghost function variable and will be eliminated by a deductive approach later.

After we have LOF, we further impose conditions (C2)–(C3) to obtain $\mathtt{L_{upd}}$ and $\mathtt{L_{leaf}}$.

$$\text{(C2)}\ \forall \mathtt{p}.\left(\mathtt{LOF(i, p[i:n])} = \mathtt{L_{upd}(i, LOF(i+1, p[i+1:n]), p[i])}\right) \quad \text{(C3)}\ \mathtt{L_{leaf}} = \mathtt{LOF(n+1, [])}$$

Specification for the MPF. Similar to LOF, we first synthesize MPF and then its initial value and updating function. Recall that if $\mathtt{MPF(i, p[1:i-1])}$ returns the same value on two subproblems, the two subproblems have the equivalent constraints. To model this property into a program synthesis task, we can equivalently rephrase this property as follows. Note that this property holds if and only if the constraint in the given CP can be equivalently transformed into a predicate that depends on the result of the MPF but not the enumerated variables for validity checking. Therefore, we impose the following condition (D1) for every instance $\mathtt{(n, C, weight, value)}$ considered in the current CEGIS iteration, and every $1 \le \mathtt{i} \le \mathtt{n}$.

$$\text{(D1)}\ \exists \odot_i\,.\left(\textstyle\sum_{j=1}^{n} \mathtt{p[j]} \cdot \mathtt{weight[j]} \le \mathtt{C} \Leftrightarrow \mathtt{MPF(i, p[1:i-1])} \odot_i \mathtt{p[i:n]}\right)$$

Similarly, we are *not* intended to synthesize the ghost function variable $\odot_i$. Furthermore, to find an efficient MA, we need to minimize the range of the MPF.

We further impose conditions (D2)–(D3) to obtain $\mathtt{M_{upd}}$ and $\mathtt{M_{init}}$.

$$\text{(D2)}\ \forall \mathtt{p}.\left(\mathtt{MPF(i+1, p[1:i])} = \mathtt{M_{upd}(i, MPF(i, p[1:i-1]), p[i])}\right) \quad \text{(D3)}\ \mathtt{M_{init}} = \mathtt{MPF(1, [])}$$

In the end, we remark that $\mathtt{n, C, weight}$ and $\mathtt{value}$ are visible to all functions to be synthesized above, but we omit this dependency for conciseness.

2.6 Solving the Synthesis Problems

Both synthesis problems for LOF and MPF are *relational* [Wang et al. 2018]. Since the number of unknown functions ($\oplus_i$ in (C1), $\odot_i$ in (D1) for each $1 \le i \le n$, the LOF and the MPF) are usually large, these two synthesis problems are beyond the reach of the previous approach on relational program synthesis [Wang et al. 2018].

Hence, we propose a novel synthesis algorithm for the two tasks. Our algorithm applies the deductive *term rewriting* [Willsey et al. 2021] to bypass the synthesis of $\odot_i$'s ($\oplus_i$'s) and reduce each task into a conventional SyGuS task. Below, we illustrate how to solve (C1)–(C3) and complete the holes $\bigcirc_{\texttt{Lleaf}}$ and $\bigcirc_{\texttt{Lupd}}$ in a single CEGIS iteration over a single instance I_0 presented in Figure 4. The procedure for (D1)–(D3) is similar, which we sketch at the end of this section.

Inductive Instantiation. We first address (C1) to obtain LOF. This specification involves ghost variables $\oplus_i$ and a complex higher-order operator Σ. We first remove Σ to facilitate the deductive transformation in the next step, which removes $\oplus_i$. Since we only need to synthesize the LOF over a set of concrete problem instances in a CEGIS iteration, we can expand the Σ operator for each problem instance. We plug in I_0 into (C1) and obtain the following.

$$\forall \mathtt{p} \quad 3 \cdot \mathtt{p}[1] + 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3] = \mathtt{LOF}(1, \mathtt{p}[1:3]) \tag{1}$$

$$\exists \oplus_2 \ \forall \mathtt{p} \quad 3 \cdot \mathtt{p}[1] + 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3] = \mathtt{p}[1:1] \oplus_2 \mathtt{LOF}(2, \mathtt{p}[2:3]) \tag{2}$$

$$\exists \oplus_3 \ \forall \mathtt{p} \quad 3 \cdot \mathtt{p}[1] + 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3] = \mathtt{p}[1:2] \oplus_3 \mathtt{LOF}(3, \mathtt{p}[3:3]) \tag{3}$$

Here all ghost function variables $\oplus_2, \oplus_3$ are monotone with respect to the second argument.

Deductive term rewriting. To remove the ghost variables $\oplus_i$, SynMem integrates a deductive term rewriting system. It systematically rewrites the LHS of (1)–(3) into the equivalent forms, implicitly exploring different candidates of $\oplus_i$.

Concretely, consider the condition (2) above first. By rewriting the LHS of (2) as $(3 \cdot \mathtt{p}[1]) + (4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3])$, we can deduce that, once we find a LOF such that $\forall \mathtt{p}[1:3]\ \mathtt{LOF}(2, \mathtt{p}[2:3]) = 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3]$, we can choose $\oplus_2$ as $3 \cdot \mathtt{p}[1] + \mathtt{LOF}(3, \mathtt{p}[2:3])$ to establish (2). Similarly, we can apply rewriting and choose $\oplus_3$ as $(3 \cdot \mathtt{p}[1] + 4 \cdot \mathtt{p}[2]) + \mathtt{LOF}(3, \mathtt{p}[3:3])$, given $\mathtt{LOF}(3, \mathtt{p}[3:3]) = 5 \cdot \mathtt{p}[3]$. Note that $\oplus_2$ and $\oplus_3$ only involve primitive operators $+$ and $\cdot$. Thus, we could trivially check the monotonicity since $a + b$ increases as b increases. In Figure 7, we present a possible resulting synthesis task for LOF after we have rewritten (1)–(3).

$$\forall \mathtt{p} \quad \mathtt{LOF}(1, \mathtt{p}[1:3]) = 3 \cdot \mathtt{p}[1] + 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3]$$
$$\forall \mathtt{p} \quad \mathtt{LOF}(2, \mathtt{p}[2:3]) = 4 \cdot \mathtt{p}[2] + 5 \cdot \mathtt{p}[3]$$
$$\forall \mathtt{p} \quad \mathtt{LOF}(3, \mathtt{p}[3:3]) = 5 \cdot \mathtt{p}[3]$$

Fig. 7. A possible condition for synthesis

Reduction to SyGuS. Figure 7 shows a conventional SyGuS specification [Alur et al. 2018]. In a SyGuS problem, we are given a domain-specific language (DSL) representing the whole program space and need to find a correct program on all inputs. Since the LOF admits an updating function, it must be a program in the structural recursion form. Thus, SynMem synthesizes the LOF under a language that includes the compositions of the typical list structural recursion operators (e.g., map, filter, sum, etc.) We use the basic bottom-up enumerative method to solve this task, synthesizing $\mathtt{LOF}(\mathtt{i}, \mathtt{p}[\mathtt{i}:\mathtt{n}]) = \mathtt{sum}\,(\mathtt{map}(\lambda \mathtt{j}.\mathtt{value}[\mathtt{j}] \cdot \mathtt{p}[\mathtt{j}], \mathtt{index}\ \mathtt{p}^{\circ})$ in our DSL, which is equivalent to $\sum_{\mathtt{j}=\mathtt{i}}^{\mathtt{n}} \mathtt{value}[\mathtt{j}] \cdot \mathtt{p}[\mathtt{j}]$.

After synthesizing the LOF, conditions (C2) and (C3) are conventional SyGuS tasks. Thus, we assume an external SyGuS solver to synthesize of the updating function. In our implementation, we carefully restrict our DSL such that for any synthesized LOF, we can use syntactic transformations to derive its updating function. In addition, the derived updating function is guaranteed to be

$O(1)$. For KP, we obtain that $\mathsf{L}_{\mathsf{upd}}(\mathsf{i}, \mathsf{L}, \mathsf{p}[\mathsf{i}]) = \mathsf{L} + \mathsf{p}[\mathsf{i}] \cdot \mathsf{value}[\mathsf{i}]$, and its initial value, $\mathsf{L}_{\mathsf{leaf}} = 0$, completing the holes $\bigcirc_{\mathsf{Lleaf}}$ and $\bigcirc_{\mathsf{Lupd}}$.

Synthesizing the MPF. The correctness condition (D1)–(D3) for the MPF is very close to (C1)–(C3). Hence, we follow the same procedure to synthesize MPF and derive its initial value and the updating function. However, there are two differences.

First, we need to synthesize functions that output a tuple of scalars. We will enumerate the number of tuples in the output and synthesize the function for each component in the tuple. We reformalize this problem as a hitting set problem and apply the pruning on the max-degree bound [Bläsius et al. 2022] to efficiently solve this problem.

Second, we need to find an MPF with a minimized output range. We observe that applying a structural recursion function (e.g., sum, max) shrinks the range of the result. Thus, the larger the synthesized program, the smaller ranges the MPF would have. Thus, we follow the heuristics that enumerates the programs from large to small for the SyGuS problem. Applying the above procedure to KP, we obtain $\mathsf{M}(i, \mathsf{p}[1 : \mathsf{i} - 1]) = \mathsf{sum}\,(\mathsf{map}(\lambda \mathsf{j}.\mathsf{weight}[\mathsf{j}] \cdot \mathsf{p}[\mathsf{j}], \mathsf{index}\ \mathsf{p}^\bullet)$, and generate $\mathsf{M}_{\mathsf{upd}}(\mathsf{i}, \mathsf{M}, \mathsf{p}[\mathsf{i}]) = \mathsf{M} + \mathsf{p}[\mathsf{i}] \cdot \mathsf{weight}[\mathsf{i}]$ and $\mathsf{M}_{\mathsf{init}} = 0$, completing the holes $\bigcirc_{\mathsf{Minit}}$ and $\bigcirc_{\mathsf{Mupd}}$.

In the end, we remark that though conditions (D1) and (D2) are syntactically similar, the synthesis of (D1) is a relational program synthesis task, but the synthesis of (D2) is a classic SyGuS task. Thus, SynMem treats differently on these two conditions. Furthermore, note that SynMem is straightforwardly *sound* due to our insights into MAs and is *complete* (the reduction from Figure 6 to Figure 7 never excludes valid programs) if the set of equivalent expressions is recursively enumerable (Section 5.2).

3 FORMALIZING THE SYNTHESIS TASK

Representing CPs. The essential attributes of our specification have been illustrated in Section 2. Below we present the language features in more detail. Please refer to the full version of this paper for a complete illustration. A typical CP specification consists of four parts as follows.

1. Inputs. This part consists of the parameters of the problem. Each parameter is specified as an atomic value or an array of atomic values. The array length may depend on other parameters. The domain of each atomic value is a bounded interval `l..r` of integers, whose endpoints `l` and `r` can also depend on other parameters. For simplicity, we only consider 1-dimensional arrays. Other data structures, such as n-dimensional arrays or lists, can be converted to 1-dimensional arrays and fit into our framework.

2. Solution. The solution part consists of k arrays $\mathsf{sol}^1, \ldots, \mathsf{sol}^k$ of atomic variables. The array sol_i has the index $\mathsf{p}_\mathsf{i}..\mathsf{q}_\mathsf{i}$ and the bounded interval $\mathsf{l}_\mathsf{i}..\mathsf{r}_\mathsf{i}$ as the domain for atomic variables, where $\mathsf{p}_\mathsf{i}, \mathsf{q}_\mathsf{i}, \mathsf{l}_\mathsf{i}$ and r_i only depend on the parameters. Still, we only consider 1-dimensional arrays.

3. Constraints. This part specifies the constraints in the solution space and defines the validity of a solution. When defining a constraint, SynMem supports common logical connectives (e.g., $\land, \lor, \neg, <, >, =$, etc.), common arithmetic operators (e.g., $+, -, \times$, max, min, etc.). SynMem also supports accumulators in the form `ACC(v in l..r)(f(v))`, where $\mathsf{ACC} \in \{$**sum**, **min**, **max**, $\ldots\}$. It iterates the fresh variable `v` from `l..r` and accumulates the results `f(v)`. Furthermore, SynMem supports for-loops to `forall(v in l..r)(expr(v))` to construct a list of constraints $[\mathsf{expr}(\mathsf{v}) \mid \mathsf{l} \le \mathsf{v} \le \mathsf{r}]$. Here, we also restrict that the range `l..r` only depends on the parameters.

4. Objective. This part specifies the type of the given CP, which has been illustrated in Section 2.1. Below, we define core concepts for the synthesis task.

Problem instances. Given the specification of a CP, the *problem instance* I is the quadruple $\langle \beta, V, C, O \rangle$ where:

- β is the assignment to all parameters in the input part, where all array lengths are consistent with the value of other parameters, and all atomic values are within their domains.
- V is the set of atomic variables for the problem instance. Given the parameters β_I, for every array $\mathtt{sol}^{\mathtt{i}}$ in the solution part, its indices $\mathtt{p_i}..\mathtt{q_i}$ and the domain of each atomic variable $\mathtt{l_i}..\mathtt{r_i}$ are fixed. Concretely, in this array, each atomic variable $\mathtt{sol}^{\mathtt{i}}\mathtt{[j]}$ ($\mathtt{p_i}(\beta) \le \mathtt{j} \le \mathtt{q_i}(\beta)$) has the domain ($\mathtt{l_i}(\beta)\,..\,\mathtt{r_i}(\beta)$). V collects the name and the domain of these variables.
- C is the set of constraints for the problem instance. This is defined as substituting the original constraint with β. We expand the for-loop `forall(v in l..r)(expr(v))` and add every `expr(v)` ($\mathtt{l}(\beta) \le \mathtt{v} \le \mathtt{r}(\beta)$) to C.
- O is the objective for the problem instance. For COPs, it is defined as substituting the original objective function with β. For CCPs and CDPs, it is a single value indicating the CP type.

In the rest of the paper, we omit the subscript I for simplicity if no confusion would be caused.

Assignments. Given a specification and its problem instance $I = \langle \beta, V, C, O \rangle$, the assignment $V^\bullet$ is a (partial) map over a subset of atomic variables. $V^\bullet$ maps each atomic variable v in this subset to a concrete value in the domain of v. The $V^\bullet$ is termed as *total* if it is a total map from $V \to \mathtt{Int}$, meaning we have fixed the values for all atomic variables. We use V^{tot} to represent a total assignment. Furthermore, for every assignment $V^\bullet$, we use $C(V^\bullet)$ ($O(V^\bullet)$, resp.) to represent the constraints (the objective, resp.) by further substituting the original one with the assignment $V^\bullet$ to part of atomic variables. Specifically, if $V^\bullet$ violates any constraint $\in C$, we set $C(V^\bullet) = \mathtt{False}$. Note that for total assignments V^{tot}, $C(V^{\text{tot}})$ is either $\mathtt{True}$ or $\mathtt{False}$, and $O(V^{\text{tot}})$ is the objective value of V^{tot} for COPs.

The synthesis goal. Given the specification, SynMem aims to synthesize a program P such that *for every parameter β*, the program P reads β and obtains the problem instance $I = \langle \beta, V, C, O \rangle$, and outputs the correct value $P(I)$ such that.

- For combinatorial optimization problems (COPs), $P(I)$ is the maximum objective value of a valid total assignment, i.e., $P(I) = \max_{V^{\text{tot}}}\{O(V^{\text{tot}}) \mid C(V^{\text{tot}}) = \mathtt{True}\}$.
- For combinatorial decision problems (CDPs), $P(I)$ is whether there exists a valid solution, i.e., $P(I) = \mathtt{True}$ iff $\{V^{\text{tot}} \mid C(V^{\text{tot}}) = \mathtt{True}\}$ is non-empty.
- For combinatorial counting problems (CCPs), $P(I)$ is the number of valid solutions, i.e., $P(I) = |\{V^{\text{tot}} \mid C(V^{\text{tot}}) = \mathtt{True}\}|$.

Specifically, SynMem aims to synthesize a MA whose structure is defined in Section 4.

4 THE STRUCTURE OF THE MEMOIZATION ALGORITHMS

In this part, we illustrate the structures of the MAs. We have illustrated the main ideas in Section 2.2. Below, we provide the formal treatment. In our structuring, the MAs are built upon the exhaustive search algorithms. It enumerates all atomic variables from the given *enumeration order*. Thus, we must first formally define the enumeration order as follows.

Enumeration order. The enumeration order specifies the order of the atomic variables for every problem instance. Different orders yield different MAs. Thus, SynMem tries all enumeration orders in the prescribed space, whose syntax and semantics are presented in Figure 8. The space consists of all permutations of the arrays in the solution part. It also supports enumerating several arrays with the same length simultaneously. Note that `zip` can have only one parameter, which means the array itself. Within each array $\mathtt{sol[p_i..q_i]}$, the program enumerates from $\mathtt{sol[p_i]}$ to $\mathtt{sol[q_i]}$. Given the problem instance I, the semantics $[\![E]\!]$ is a list, where the i-th element stores the name and the domain for the set of atomic variables enumerated at the i-th step.

$$\begin{array}{lcll} \mathtt{E} & ::= & \mathtt{zip}(\mathtt{sol}^1, \ldots, \mathtt{sol}^t) & \text{where } \mathtt{sol}^1, \ldots \mathtt{sol}^t \text{ have the same length.} \\ & \mid & \mathtt{E}_1 ; \mathtt{E}_2 & \end{array}$$

$$[\![\mathtt{E}_1;\mathtt{E}_2]\!] = [\![\mathtt{E}_1]\!] +\!\!+ [\![\mathtt{E}_2]\!]$$

$$[\![\mathtt{zip}(\mathtt{sol}^1,\ldots,\mathtt{sol}^t)]\!] = [(\mathtt{sol}^1[\mathtt{l}_1],\ldots,\mathtt{sol}^t[\mathtt{l}_t]),\ldots,(\mathtt{sol}^1[\mathtt{r}_1],\ldots,\mathtt{sol}^t[\mathtt{r}_t])]$$

Fig. 8. The syntax and the semantics of enumeration orders

Example 4.1. Suppose there are two arrays of atomic variables a and b with the same index 1..n. Then, after assigning the a value to n, zip(a); zip(b) evaluates to the order [a[1], . . . , a[n], b[1], . . . , b[n]] that sequentially enumerates two arrays a and b, but zip(a, b) evaluates to the order [(a[1], b[1]), . . . , (a[n], b[n])] that enumerates (a[i], b[i]) at the same time.

We only consider enumeration orders E such that all arrays in the solution part appear exactly once. Such orders visit all atomic variables exactly once for every problem instance. Given the order E and a problem instance I, we can rearrange the atomic variables V into a list $[V_1, \ldots, V_m]$, where each V_i is the set of atomic variables to be enumerated at the i-th step.

Below, we present the template for the exhaustive search and its sketch. We fix the enumeration order E, the concrete instance $I = \langle \beta, V, C, O \rangle$.

4.1 The Exhaustive Search and Its Sketch

In our structures, the exhaustive search and its search sketch follow a *template* as follows.

Exhaustive search. The template for the exhaustive search is presented in the left half of Figure 9. It follows the order $[\![E]\!]$ (denoted by $\mathtt{V} = [\mathtt{V}_1, \ldots, \mathtt{V}_m]$) of the atomic variables on the problem instance I and recursively enumerates each $\mathtt{V}_i$ in the procedure Search.

Consider the invocation search(i, $V^\bullet$), where i is the *search stage*, meaning that the exhaustive search will currently enumerate the atomic variables $V[\mathtt{i}]$, and $V^\bullet$ is the partial assignment to the enumerated atomic variables $V[1 : \mathtt{i} - 1]$. Whenever the assignment $V^\bullet$ is invalid, it returns the invalid result obj_invalid immediately (Lines 5–6). After enumerating all variables, it returns the objective value obj_val($V^\bullet$) for the total assignment $V^\bullet$ (Lines 7–8). It enumerates the assignment to $V[\mathtt{i}]$ and merges the result of all sub-procedure calls using the function obj_merge. The type of the given CP entirely determines the expressions obj_invalid, obj_val($V^\bullet$), and obj_merge, which is listed in Figure 9. Since the enumeration order visits each atomic variable exactly once, it is easy to see that the following theorem holds.

Theorem 4.2 (Soundness of the exhaustive search). *Given any specification and any enumeration order E, the exhaustive search algorithm (Figure 9) matches the synthesis goal in Section 3.*

Subproblem. Given the enumeration order E and the problem instance I, each invocation search(i, $V^\bullet$) can be identified as an intermediate subproblem $Q(\mathtt{i}, V^\bullet)$, whose specification is as follows.

- The atomic variables of the subproblem is $V[\mathtt{i} : \mathtt{m}]$, which is the set of unknown atomic variables at the search stage i.
- The constraint of this subproblem is $C(V^\bullet)$
- The objective of this subproblem is $O(V^\bullet)$.

Search sketch. As discussed in Section 2.2, the subproblems in the exhaustive search does not enable efficient reusing. To optimize the search, the MAs introduces the local objective function (LOF) and

```
const enum_order $E$;
parameter $\beta$;
problem_instance $I$;
atom_var_info[] V_info;
void search(int i,agn_t $V^\bullet$) {
  if($C(V^\bullet)$==False)
    return obj_invalid;
  if (i>length(V_info))
    return obj_val($V^\bullet$)
  int ans=obj_invalid;
  for $V_i^\bullet \in$ V_info[i].dom
    ans=obj_merge(ans,subval);
    where subval=search(i+1,$V_2^\bullet$)
    and $V_2^\bullet = V^\bullet +\!\!+ V_i^\bullet$
  return ans;
}
int main() {
  read(&$\beta$);
  $I$=gen_instance($\beta$);
  V = $[\![E]\!]$;
  return search(1,[]);
}
```

	COP	CDP	CCP
obj_invalid	$-\infty$	False	0
obj_val($V^\bullet$)	$O(V^\bullet)$	True	1
obj_merge(a, b)	$\max(a, b)$	$a \vee b$	$a + b$

	CDP	CCP
$\bigcirc_{\texttt{Lleaf}}$	True	1
$\bigcirc_{\texttt{Lupd}}$	L	L

```
const enum_order $E$;
parameter $\beta$;
problem_instance $I$;
atom_var_info[] V_info;
map<(int, int$^k$), int> mem;
int L$_{\text{leaf}}$ = $\bigcirc_{\texttt{Lleaf}}$, M$_{\text{init}}$ = $\bigcirc_{\texttt{Minit}}$;
int L$_{\text{upd}}$(int i,int L,agn_t $V_i^\bullet$)
  { return $\bigcirc_{\texttt{Lupd}}$;}
int$^k$ M$_{\text{upd}}$(int i,int$^k$ M,agn_t $V_i^\bullet$)
  { return $\bigcirc_{\texttt{Mupd}}$;}

int search(int i,int$^k$ M,agn_t $V^\bullet$) {
  if ($C(V^\bullet)$==False)
    return obj_invalid;
  if (i>length(V_info))
    return L$_{\text{leaf}}$;
  if(mem.find(i,M)!=mem.end())
    return mem[(i,M)];

  ans=obj_invalid;
  for $V_i^\bullet \in$ V_info[i].dom
    ans=obj_merge(ans,L$_{\text{upd}}$(i,L,$V_i^\bullet$));
    where L=search(i+1,M',$V_2^\bullet$)
    and M'=M$_{\text{upd}}$(i,M,$V_i^\bullet$)
    and $V_2^\bullet = V^\bullet +\!\!+ V_i^\bullet$
  return mem[(i,M)]=ans;
}
int main() {
  read(&$\beta$);
  I = gen_instance($\beta$);
  V = $[\![E]\!]$;
  return search(1,M$_{\text{init}}$,[]);
}
```

Fig. 9. The pseudo-code for the template for the exhaustive search (left) and its sketch (right). The holes $\bigcirc_{\texttt{Lleaf}}$, $\bigcirc_{\texttt{Minit}}$, $\bigcirc_{\texttt{Lupd}}$, $\bigcirc_{\texttt{Mupd}}$ need to be filled by synthesized expressions.

the memoization partition function (MPF). Thus, we define the template for the search sketch (Figure 9), which follows a similar recursive procedure with the exhaustive search but adds some codes and holes for the LOF and the MPF, which we illustrate in Sections 4.2 and 4.3.

4.2 Local Objective Functions

We introduce the LOF and its updating function in our structure of MAs to make the objective function of different subproblems more likely to be equivalent. For CDPs and CCPs, since there is no objective function, the holes $\bigcirc_{\texttt{Lleaf}}$ and $\bigcirc_{\texttt{Lupd}}$ can be trivially filled, as shown in Figure 9. Below, we only discuss the case for COPs. We present the formalization of LOF and its updating function.

Local Objective Function (LOF). The $\texttt{LOF}(\texttt{i}, V[\texttt{i} : \texttt{m}])$ is a function independent with the set of enumerated atomic variables. For each subproblem $Q(\texttt{i}, V^\bullet)$, we replace its objective function from

$O(V^\bullet)$ with LOF. We generalize the condition (C1) in Section 2.5 and define the LOF as follows.

$$\forall I.\forall \mathtt{i} \in [1 : \mathtt{m}].\exists \oplus_i .\forall V^\bullet.\forall V^\circ.O(V^\bullet +\!\!+ V^\circ) = V^\bullet \oplus_i \mathsf{LOF}(i, V^\circ) \tag{4}$$

where we further restrict that each $\oplus_i$ is monotonically increasing with respect to its second argument x. Moreover, for $i = 1$, we restrict that $\forall x.[] \oplus_1 x = x$.

Updating function of the LOF. We also need the updating function $\mathsf{L_{upd}}(\mathtt{i}, \mathsf{L}, V_{\mathtt{i}}^\bullet)$ to obtain the LOF from those of the child subproblems, where the first parameter is the search stage, the second parameter is the LOF of the child subproblem, the third parameter is the assignment to the currently enumerated atomic variables $V[\mathtt{i}]$. It must satisfy the following equation.

$$\forall I.\forall \mathtt{i} \in [1 : \mathtt{m}].\forall V^{\text{tot}}.\mathsf{LOF}(\mathtt{i}, V^{\text{tot}}[\mathtt{i} : \mathtt{m}]) = \mathsf{L_{upd}}(\mathtt{i}, \mathsf{LOF}(\mathtt{i} + 1, V^{\text{tot}}[\mathtt{i} + 1 : \mathtt{m}]), V^{\text{tot}}[i]). \tag{5}$$

In the template of the search sketch, for the LOF, we leave two holes:

- $\bigcirc_{\mathtt{Lleaf}}$ that equals the LOF for the leaf subproblems $\mathsf{LOF}(\mathtt{m} + 1, [])$, and is applied in Line 16.
- $\bigcirc_{\mathsf{Lupd}}$ that equals the updating function $\mathsf{L_{upd}}(\mathtt{i}, \mathsf{L}, V_{\mathtt{i}}^\bullet)$, and is applied in Line 22.

4.3 Memoization Partition Functions

We introduce the MPF and its updating function to quickly identify whether two subproblems have the same valid solution set.

Memoization Partition Function (MPF). The $\mathsf{MPF}(\mathtt{i}, V[1 : \mathtt{i} - 1])$ is a function over the enumerated variables $V[1 : \mathtt{i} - 1]$ and the parameters in the original specification. It outputs a tuple of scalars. We generalize the condition (D1) (Section 2.5) and define the MPF as follows.

$$\forall I.\forall C_0 \in C.\forall \mathtt{i} \in [1 : \mathtt{m}].\exists \odot_{C_0,i} .\forall V^\bullet.\forall V^\circ. \left(C_0 \Leftrightarrow \mathsf{MPF}(i, V^\bullet) \odot_{C_0,i} V^\circ\right) \tag{6}$$

Updating function of the MPF. We need an updating function $\mathsf{M_{upd}}(\mathtt{i}, \mathsf{M}, \mathsf{V}_{\mathtt{i}}^\bullet)$, whose first parameter is the search stage i, second parameter is the MPF value of the parent subproblem and the third parameter is the assignment to currently enumerated variables $V[\mathtt{i}]$.

$$\forall I.\forall \mathtt{i} \in [1 : \mathtt{m}].\forall V^{\text{tot}}.\mathsf{MPF}(i + 1, V^{\text{tot}}[1 : i]) = \mathsf{M_{upd}}(i, \mathsf{MPF}(i, V^{\text{tot}}[1 : i - 1]), V^{\text{tot}}[i]). \tag{7}$$

In the template (Figure 9), we add the parameter `W` in the procedure `search` to track the MPF value, we also leave two holes:

- $\bigcirc_{\mathtt{Minit}}$ that equals the initial value for $\mathsf{MPF}(1, [])$ and is applied in Line 32.
- $\bigcirc_{\mathsf{Mupd}}$ that equals the updating function for the MPF and is applied in Line 24.

In the end, we remark the input parameters are visible to all functions above. We formalize the properties of our structuring as follows.

Theorem 4.3 (Correctness and Efficiency of Our Structure). *Given any specification, if there are functions* $\mathsf{LOF}, \mathsf{L_{upd}}, \mathsf{MPF}, \mathsf{M_{upd}}$ *satisfying the conditions (4)–(7), then we can fill the holes* $\bigcirc_{\mathtt{Lleaf}}, \bigcirc_{\mathsf{Lupd}}, \bigcirc_{\mathtt{Minit}}, \bigcirc_{\mathsf{Mupd}}$ *in the search sketch (Figure 9), deriving the MA* P_{mem} *such that:*

- *(Correctness)* P_{mem} *is equivalent to the exhaustive search algorithm in Figure 9, thus matches the synthesis goal in Section 3.*
- *(Efficiency) Given any problem instance* $I = \langle \beta, V, C, O \rangle$, *the number of equivalence classes of the subproblems is* $O(|V| \cdot \text{range})$, *where* range *the number of different outputs of the MPF on the problem instance* I.

Proof. For the correctness part, consider two invocations of the procedure `search` that correspond to two subproblems $Q(\mathtt{i}, V_1^\bullet)$ and $Q(\mathtt{i}, V_2^\bullet)$. Suppose they have the same `i` and the same MPF value $\mathsf{MPF}(\mathtt{i}, V_1^\bullet) = \mathsf{MPF}(\mathtt{i}, V_2^\bullet)$. In that case, they have the same set of unknown atomic variables $V[\mathtt{i} : \mathtt{m}]$, the same LOF, and the same valid solution set over $V[\mathtt{i} : \mathtt{m}]$. Hence, the output of the two invocations must be the same. As a result, we can set up a map `mem` to safely reuse between these subproblems. The efficiency part is straightforward. □

5 SYNTHESIZING MEMOIZATION ALGORITHMS

5.1 The Outmost Controller

In the outmost iteration (Algorithm 1), SynMem tries enumeration orders from the space (Figure 8) since different enumeration orders yield different MAs. SynMem also iteratively increments the hyperparameters B and K to bound the program space when synthesizing the LOF and the MPF, which will discuss in detail in Section 5.2. If the synthesis fails, it increments these parameters and restarts a new iteration. It relies on two procedures `GenSketch` and `CEGIS`, illustrated as follows.

`GenSketch`. Given a specification and the enumeration order E, this procedure generates the exhaustive search and its sketch from the template (Figure 9). Note that the length of the arrays in the solution part, the domain of each atomic variable, and all range expressions in the constraint and objection part only depend on the input parameters, and are fixed given the parameter choice β_I. As a result, we can easily deduce the program that takes β_I as input and outputs other components S_I, C_I, and O_I in the program instance, completing the generating procedure.

`CEGIS`. To tackle the sketch problem from a reference exhaustive search implementation, SynMem applies the counter-example guided inductive program synthesis framework (Section 2.5) to reduce the original synthesis problem to synthesizing an efficient and correct MA that is equivalent to the exhaustive search for a finite set of problem instances $\mathcal{E}$ (Algorithm 2).

Algorithm 1: The Outmost Controller

```
Input: The specification
Output: The memoization program M
B ← $B_{init}$; K ← $K_{init}$
while not synthesized yet do
    foreach $E \in$ GetEnumOrder($\mathcal{S}$) do
        $M_0$ ← GenSketch($E$)
        M ← CEGIS(B, K, $E$, $M_0$)
        if M ≠ fail then return $P$;
    end
    B ← B + $B_{step}$
    K ← K + $K_{step}$
end
```

Algorithm 2: The CEGIS Part

```
Input: Max AST size B. Max #tuples in the MPF K.
       Enumeration order $E$. Search sketch $P_0$.
Output: The memoization program P
$\mathcal{E} \leftarrow \emptyset$;  P ← Error
while M ≠ fail ∧ ¬Verify(P) do
    $I_0$ ← CounterEx(M)
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{I_0\}$
    $\bigcirc_{Lleaf}, \bigcirc_{Lupd}$ ← SynLOF(B, $E$, $\mathcal{E}$)
    $\bigcirc_{Minit}, \bigcirc_{Mupd}$ ← SynMPF(B, K, $E$, $\mathcal{E}$)
    if none $\bigcirc_{Lleaf}, \bigcirc_{Lupd}, \bigcirc_{Minit}, \bigcirc_{Mupd}$ fails then
        M ← Plug the holes above into $M_0$
    end
end
return M
```

5.2 Inductive Synthesis of the LOF and the MPF

In each CEGIS iteration, we complete the search sketch (Figure 9). Instead of applying the general sketch method, we apply Theorem 4.3, which reduces the sketch problem to synthesizing the LOF, MPF, and their updating functions with respect to the conditions (4)–(7). These conditions yield two *independent* tasks on quantified relational program synthesis, enabling an efficient synthesis procedure. Furthermore, we can control the efficiency of the synthesized MA by minimizing the range of the MPF. Below, we present the synthesis of the MPF. After synthesizing the MPF, the synthesis of its updating function (Condition (7)) is a conventional SyGuS problem. We assume an external solver $\mathcal{U}$ to solve this synthesis task. The synthesis of the LOF (and its updating function) follows a similar procedure, which we sketch at the end of this section. Below, we first present the domain specific language (DSL) for `LOF`, `MPF`.

DSL Description. The DSL consists of constants, all input parameters, the search stage `i`, and the unknown suffix (enumerated prefix, resp.) of each array $\mathtt{sol}^{\mathtt{i}}$ in the solution part. Since the LOF and the MPF admit updating functions, they must be programs represented by structural recursions. Thus, the DSL consists of the compositions of common structural recursion operators, such as `map`, `filter`, `sum`, `max`, `min`, `length` and `suffix`, etc. It also consists of primitive operators

$+, -, \times$, logical connectives $\wedge, \vee, \rightarrow$ and the array access operator `access`. However, we forbid the occurrence of the structural recursion operators and the variable `i` in the grammar of the function part for higher-order operators.

***Synthesis of the MPF* SynMPF.** SynMem synthesizes the MPF from its formalization (6). We present the pseudo-code of this procedure in Algorithm 3. Since in each CEGIS iteration, SynMem only aims to synthesize an MA equivalent to the exhaustive search on a set of problem instances $\mathcal{E}$, SynMem changes (6) by replacing the bound of the problem instance I from all instances to $\mathcal{E}$.

Recall that the output of the MPF is a tuple of scalars. Thus, SynMem treats synthesizing the MPF as the joint synthesis of ℓ programs $\mathtt{M}_1, \ldots \mathtt{M}_\ell$, where ℓ is the number of the components in the tuple, and $\mathtt{M_j}$ is the j-th component. SynMem applies two hyperparameters K and B to bound the program space, where K upper bounds ℓ and B upper bounds the number of AST nodes for each $\mathtt{M_i}$. The synthesis of the MPF consists of the following steps.

Algorithm 3: The procedure SynMPF

```
Input: Max AST size B. Max #components K. Max #Trials Lim. Enum order $E$. Set of instances $\mathcal{E}$.
Output: The expressions for the holes $\bigcirc_{\mathtt{Minit}}$ and $\bigcirc_{\mathtt{Mupd}}$, or fail for the failure
//Step 1
$\Phi_s \leftarrow \emptyset$
foreach $I \in \mathcal{E}, 1 \le i \le m, C_0 \in C_I$ do
    Collect the constraint as in (8) into $\Phi_s$
end
for $t \le$ Lim $\wedge$ rewriting not exhausted do
    // Step 2
    $\Psi_s \leftarrow \emptyset$
    foreach $\Phi(I, C_0, i) \in \Phi_s$ do
        $R(c_1^\bullet, \ldots, c_p^\bullet, c_1^\circ, \ldots, c_q^\circ) \in (\mathtt{Check} \circ \mathtt{Rewrite})(C_0)$
        foreach $1 \le j \le p$ do
            Collect the constraints as in (9) into $\Psi_s$
        end
    end
    // Step 3
    $\mathtt{M}_1, \ldots \mathtt{M}_\ell \leftarrow \mathtt{MultiSynth}(\mathtt{B}, \mathtt{K}, \Psi_s)$
    if $\mathtt{M}_1, \ldots \mathtt{M}_\ell$ does not fail then
        $\mathtt{M} \leftarrow (\mathtt{M}_1, \ldots \mathtt{M}_\ell)$
        $\bigcirc_{\mathtt{Mupd}} \leftarrow \mathcal{U}(\mathtt{M})$
        $\bigcirc_{\mathtt{Minit}} \leftarrow \mathtt{M}(1, [\,])$
        return $\bigcirc_{\mathtt{Minit}}, \bigcirc_{\mathtt{Mupd}}$
    end
end
return fail
```

Step 1: Inductive Instantiation. SynMem scans every instance $I = (\beta_I, V_I, C_I, O_I) \in \mathcal{E}$ and calculates the order $[V_1, \ldots, V_m] := [\![E]\!]_I$. Then, for every $1 \le \mathtt{i} \le \mathtt{m}$ and every $C_0 \in C_I$, SynMem instantiates the condition (6) into $\Phi(I, C_0, i)$ as follows, and collects this condition.

$$\Phi(I, C_0, i) := \left(\exists \odot_{C_0,i} .\forall V^\bullet \text{ over } V_I[1 : \mathtt{i} - 1].\forall V^\circ \text{ over } V_I[\mathtt{i} : \mathtt{m}].\left(C_0 \Leftrightarrow \mathtt{MPF}(i, V^\bullet) \odot_{C_0,i} V^\circ\right)\right) \tag{8}$$

Step 2: Deductive term rewriting. In this step, SynMem further applies a reduction to each collected condition $\Phi(I, C_0, i)$ to eliminate the quantifier $\odot_{C_0,i}$. The reduction is accomplished via the deductive term rewriting, which searches an equivalent expression of C_0. SynMem assumes the existence of a term rewriting procedure $\mathtt{Rewrite}(C_0)$ which can be implemented by a rewriting system suitable for the target DSL [Brillout et al. 2011; Marché 1996; Marché and Urbain 1998; Willsey et al. 2021]. Whenever SynMem invokes $\mathtt{Rewrite}(C_0)$, it either returns "`exhausted`" meaning the exhaustion of equivalent forms, or a new equivalent expression of C_0. SynMem then applies the procedure `Check` to check if the rewriting result is in the form $R(c_1^\bullet, \ldots, c_p^\bullet, c_1^\circ, \ldots, c_q^\circ)$, where each subexpression $c_j^\bullet$ only consists of $V_I[1 : \mathtt{i} - 1]$, and each c_j° only consists of $V_I[\mathtt{i} : \mathtt{m}]$. After obtaining such a form, we can reduce the constraint $\Phi(I, C_0, i)$ into p conditions $\Psi(I, c^\bullet_1, i), \dots \Psi(I, c^\bullet_p, i)$, where each $\Psi(I, c^\bullet_j, i)$ is defined as follows, eliminating the quantifier $\odot_{C_0,i}$:

$$\Psi(I, c^\bullet_j, i) := \left(\exists t \in [1 : \ell].\forall V^\bullet.\left(c^\bullet_j(V^\bullet) = \mathsf{M}_t(i, V^\bullet)\right)\right) \tag{9}$$

Intuitively, if every $c^\bullet_j$ equals a component of the MPF $\mathsf{MPF}_{t_j}(i, V^\bullet)$ for some t_j, then we can choose $\odot_{C_0,i}$ as $R(\mathsf{M}_{t_1}(i, V^\bullet), \dots, \mathsf{M}_{t_p}(i, V^\bullet), c^\circ_1, \dots, c^\circ_q)$ to satisfy (8). Below, we show that the reduction in this step is *sound and complete*, assuming an ideal term rewriting system.

Theorem 5.1. *If the MPF* M *satisfies* $\Phi(I, C_0, i)$*, then there exists an equivalent expression of* C_0 *in the form* $R(c^\bullet_1, \dots, c^\bullet_p, c^\circ_1, \dots, c^\circ_q)$ *as above such that* M *satifies (9), and vice versa.*

Proof. For the "then" side, suppose (8) holds. Then, the truth value of C_0 is a function on M and V°. Consider the abstract syntax tree of this function, whose leaf node is either a component of M or an atomic variable in V°. Thus, this function is in the form $R(c^\bullet_1, \dots, c^\bullet_p, c^\circ_1, \dots, c^\circ_q)$ as above and M satifies (9). For the "vice versa" side, we follow the intuition above. □

Step 3: Inductive synthesis of (9). After the reduction above, the goal becomes finding $\leq \mathsf{K}$ programs to satisfy all reduced conditions in the form $\Psi(I, c^\bullet, i)$ collecting in Step 2. SynMem invokes the procedure `MultiSynth` to solve this problem. The detail of this procedure is as follows.

This procedure recursively enumerates $\mathsf{M}_1, \mathsf{M}_2, \dots$ from the space of programs with the number of AST nodes $\leq \mathsf{B}$. It returns if there have been more than K programs or the current programs $\mathsf{M}_1, \dots, \mathsf{M}_\ell$ have satisfied all reduced conditions in the form $\Psi(I, c^\bullet, i)$, which could be easily checked since the domains of t and $V^\bullet$ are finite for every fixed problem instance I.

To obtain the MPF with a small range, SynMem applies a lightweight heuristic search specific to the DSL above to synthesize each component in the MPF with a small range. Note that applying a structural recursion operator (e.g., `sum`, `max`) in our DSL shrinks the range of the result. Thus, the larger the synthesized program, the smaller ranges the MPF would have. Thus, SynMem follows the heuristics that enumerates the programs (with AST size $\leq \mathsf{B}$) from large to small in the procedure `MultiSynth` and outputs the first successful result.

However, the procedure above is too slow. We can prune the synthesis procedure by reformulating the synthesis task in this step. We say that a program M^* *hits* the condition $\Psi(I, c^\bullet, i)$ if $\forall V^\bullet.c^\bullet(V^\bullet) = M^*(V^\bullet)$. In other words, the condition $\Psi(I, c^\bullet, i)$ will be satisfied if we add M^* to a component of the MPF. From this perspective, the goal of this step becomes finding $\leq \mathsf{K}$ programs that hit all conditions in the form $\Psi(I, c^\bullet, i)$. Then, we apply the pruning based on the maximum degree bound proposition [Bläsius et al. 2022], which is given below.

Proposition 5.2. *If there are* $\leq \mathsf{K}$ *programs* $\mathsf{M}_1, \dots, \mathsf{M}_{\leq \mathsf{K}}$ *that satisfy all conditions, then there exists some* $\mathsf{M}_\mathtt{i}$ *among them that covers* $\geq \frac{1}{\mathsf{K}}$ *fraction of the conditions.*

Proof. Assume that each $p \in \mathcal{L}$ covers $< \frac{1}{\mathsf{K}}$ fraction of the conditions, then for all m programs $p_1, \cdots, p_m \in \mathcal{L}(m \leq \mathsf{K})$, they can cover $< \frac{m}{\mathsf{K}} \leq 1$ fraction of the conditions at most, which contradicts the fact that exists $\leq \mathsf{K}$ programs can cover all conditions. □

By the proposition above, when enumerating the program $\mathsf{M}_\mathtt{i}$, we only consider the programs with AST size $\leq \mathsf{B}$ and hits $\geq \frac{1}{\mathsf{K}-\mathtt{i}+1}$ fraction of conditions, which greatly excludes invalid programs and significantly speeds up the synthesis procedure.

If SynMem successfully finds $\leq \mathsf{K}$ programs that satisfy all conditions with the minimized range, then SynMem finds the MPF M by tupling these programs together. Then, SynMem invokes the external synthesizer $\mathcal{U}$ to synthesize the updating function for the MPF to fill the hole $\bigcirc_{\mathsf{Mupd}}$, and evaluates $\mathsf{M}(1, [])$ to obtain the initial value for the MPF to fill the hole $\bigcirc_{\mathtt{Minit}}$.

Backtracking. However, if the synthesis in Step 3 fails to find $\leq$ K programs that satisfy all conditions, then SynMem backtracks to Step 2 to search for another term rewriting result, and starts Step 3 again. Suppose the number of failing trails in Step 3 exceeds a prescribed number Lim or the term rewriting result is exhausted. In that case, we exit Step 2 and report failure to synthesize the programs for the holes $\bigcirc_{\mathtt{Minit}}$ and $\bigcirc_{\mathtt{Mupd}}$.

***Synthesis of the optimal substructures* SynLOF**. Below, we present SynLOF, which synthesizes the LOF and its updating function. Note that for CCPs and CDPs, this procedure is trivial since there is no optimization goal. We list the synthesis result for CCPs and CDPs in the template of the search sketch (Figure 9). For COPs, SynLOF follows almost the same procedure as SynMPF (Algorithm 3). Hence, we only illustrate the differences.

Step 1. In this step, SynMem collects conditions for the LOF following the condition (4). (Line 4)

Step 2. In this step, SynMem rewrites the conditions collected in Step 1 into the form $\oplus(c^\circ, c_1^\bullet, \ldots, c_p^\bullet)$ (Line 10), where c° only depends on the unknown atomic variables $V[\mathtt{i:m}]$, and every $c_i^\bullet$ only depends on the enumerated atomic variables $V[\mathtt{1:i-1}]$. Furthermore, SynMem also checks whether: (i) For $i = 1$, $\forall x.[] \oplus x = x$, and (ii) the combinator $\oplus$ is monotonically increasing with respect to c°. Checking the first condition is trivial, and the second condition is checked by validating the following formula.

$$\forall V_1^\circ, V_2^\circ \text{ over } V[\mathtt{i:m}].\forall V^\bullet \text{ over } V[\mathtt{1:i-1}].\left(c^\circ(V_1^\circ) \leq c^\circ(V_2^\circ)\right) \Rightarrow \tag{10}$$

$$\left(\oplus(c^\circ(V_1^\circ), c_1^\bullet(V^\bullet), \ldots, c_p^\bullet(V^\bullet)) \leq \oplus(c^\circ(V_2^\circ), c_1^\bullet(V^\bullet), \ldots, c_p^\bullet(V^\bullet))\right) \tag{11}$$

The checking is easy since for every problem instance I, the choice of V_1°, V_2° and $V^\bullet$ is finite.

Step 3. Next, since SynMem only needs to synthesize only one LOF, we set K = 1 in this step.

In the end, we discuss the properties of our algorithm.

Soundness. Note that all steps 1–3 above are sound. Thus, given a sound and complete CEGIS verifier and a sound external synthesizer $\mathcal{U}$ for synthesizing the updating function, if SynMem successfully synthesizes the LOF, MPF, and their updating functions, then these functions satisfy all conditions (4)–(7). Furthermore, we can plug these functions into the search sketch, deriving a correct and efficient MA.

Completeness. As for the completeness, first note that by Theorem 5.1, Step 2 above is complete as long as the set of equivalent forms of C_0 is recursively enumerable, which means there is an algorithm that enumerates all equivalent forms of a given expression. Such an algorithm implements an ideal term rewriter for the procedure Rewrite(C_0). Moreover, the procedure MultiSynth is complete since it enumerates all possible combinations of programs, and the pruning by Proposition 5.2 preserves completeness. Hence, our algorithm is complete, in the sense that it never excludes valid MPFs and LOFs, as long as (1) the CEGIS verifier is sound and complete; (2) the underlying DSL for enumerating the program is expressive enough, (3) the external synthesizer $\mathcal{U}$ is complete, and (4) the set of equivalent expressions is recursively enumerable.

We remark that finding a sound and complete external synthesizer $\mathcal{U}$ is easy, which simply enumerates every program with AST size $\leq$ B. As for the recursively enumerable condition, we present a case study as follows.

Case study. Consider a CP specification as follows. We disallow multiplications and divisions across variables and allow the widely-used (i) primitive operators +, $\times$, max, and min; (ii) logical connectives $\leq, \geq, \neq, \wedge, \vee$, and $\neg$; (iii) recursive operators sum, product, max, min and forall. In this case, the constraints and the objective function are CLIA formulas over variables under a given instance. Thus, we can encode them as an expression in the Presburger arithmetics [Presburger 1931] with uninterpreted functions (if we treat an array as an uninterpreted function [Bradley et al. 2006]), which is a decidable logic [Shostak 1979]. Hence, all equivalent forms are recursively enumerable. We found that 38/40(95%) of our benchmarks fall into this case.

5.3 Optimizations

Besides the pruning in Proposition 5.2, SynMem also applies other optimizations to speed up the synthesis procedure.

Filtering out unnecessary conditions. In SynMPF, SynMem filters out constraints $C_0 \in C_I$ whose variables $\mathrm{var}(C_0)$ is a subset of $V[1 : i-1]$ or $V[i : m]$ to simplify the specification. This does not affect the correctness of the MPF because if $\mathrm{var}(C_0) \subseteq V[1 : \mathtt{i}-1]$, then C_0 is a constant after enumerating $V[1 : \mathtt{i}-1]$. Thus, there is no need to consider this constant constraint. On the other hand, if $\mathrm{var}(C_0) \subseteq V[\mathtt{i} : \mathtt{m}]$, then C_0 is fully determined by the unknown variables $V[\mathtt{i} : \mathtt{m}]$. We can trivially choose $\odot_{C_0,i}$ as C_0 itself.

Pruning invalid rewriting. In Step 2 of SynMPF, consider the subexpressions $c_1^\bullet, \dots, c_p^\bullet$ in the rewriting result (Line 10). If there are more than K semantically different subexpressions (which could be checked by evaluating these subexpressions over choices of $V^\bullet$), then it is impossible to use $\leq$ K functions to hit all conditions $\Psi(I, c_1^\bullet, i), \dots, \Psi(I, c_p^\bullet, i)$. Thus, we can safely discard this rewriting result and try the next rewriting. The same pruning holds for the procedure SynLOF.

Lightweight monotonicity checking. In Step 2 of SynLOF, SynMem rewrites each collected constraint in Step 1 into the form $\oplus(c^\circ, c_1^\bullet, \dots, c_p^\bullet)$ and checks whether $\oplus$ is monotonically increasing with respect to c°. This could be achieved by scanning over $V_1^\circ, V_2^\circ, V^\bullet$ as in (11), which might be costly. Thus, SynMem applies a syntactical checking in advance. Consider the abstract syntax tree of $\oplus$, SynMem extracts the path from c° to the root node and checks if every primitive operator in this path is monotone (e.g., +, min, max, etc.). If the lightweight monotonicity checking does not apply, SynMem invokes the original checking procedure.

5.4 Implementation

In this part, we present the details of the implementation of our algorithm.

Hyperparameters. We first list the hyperparameters: $\mathtt{B}_{\mathtt{init}} = 7, \mathtt{B}_{\mathtt{step}} = 2$,and $\mathtt{K}_{\mathtt{init}} = \mathtt{K}_{\mathtt{step}} = 1$. We also set $\mathtt{Lim} = 10$.

CEGIS. It is a highly non-trivial task to automatically verify the correctness of a MA on all problem instances. Thus, we consider the bounded testing method [Lee and Cho 2023; Miltner et al. 2022] that verifies the synthesized program with instances that fall into a prespecified range. In our implementation, we consider (randomly generated) 100 instances where both arrays of length and each input component fall into the interval $[1, 5]$. If scalable verification algorithms are developed in the future, we can also use these algorithms in the CEGIS part.

Term rewriting. The rewriting is an intricate procedure whose performance depends on the operators and syntactical structures of the given expression and the algebraic rules applied for term rewriting. In SynMem, after applying CEGIS, the constraints and the objective function only consists of primitive operators (e.g., $+, \times$, max, min, etc.). Hence, SynMem applies basic algebraic rules for the associativity, commutativity, and distributivity between primitive operators so that SynMem can perform each rewriting step efficiently. We apply the breadth-first search for the term rewriting procedure.

External updating function synthesizer $\mathcal{U}$. We design a sound and complete $\mathcal{U}$ specific to the DSL above. Since the DSL for LOF and MPF only consists of the compositions of structural recursions, the updating function for LOF and MPF could be generated syntactically due to the restrictions of the DSL above. For example, consider generating the updating function for $\mathtt{sum}(\mathtt{filter}(\varphi, a))$.

Suppose we update a by $a \mathbin{+\!\!+} x$, then follow the syntactic structure, we have that $\texttt{filter}(\varphi, a \mathbin{+\!\!+} x) = \texttt{filter}(\varphi, a) \mathbin{+\!\!+} \texttt{ite}(\varphi(x), [x], \texttt{nil})$, $\texttt{sum}(\texttt{filter}(\varphi, a \mathbin{+\!\!+} x)) = \texttt{sum}(\texttt{filter}(\varphi, a)) + \texttt{ite}(\varphi(x), x, 0)$. Thus, the updating function is $\texttt{orig} + (\texttt{ite}(\varphi(x), x, 0))$, where $\texttt{orig}$ represents the original output before we update a by $a \mathbin{+\!\!+} x$. Note that by the DSL construction, the synthesized updating functions are always $O(1)$.

6 EVALUATION

In this part, we evaluate our approach against the baseline Sketch [Solar-Lezama et al. 2006].

***Dataset*.** We collect CPs from Leetcode [lee [n. d.]], National Olympiad in Informatics in Provinces-Junior (a national-wide programming contest in China) [NOI [n. d.]] and previous approach [Pu et al. 2011]. We formalize these problems into our specification form. In detail, for Leetcode, we consider problems tagged with "dynamic programming" with the highest frequencies. Leetcode maintains the frequency statistics for each problem to represent the probability that this problem appears in a real-world interview. For National Olympiad in Informatics in Provinces-Junior, we consider dynamic programming tasks (tagged by ICPC gold medal winners) in the past ten years. For benchmarks in the previous approach [Pu et al. 2011], we collect CPs that are not included by Leetcode and the algorithmic contest. We exclude problems that are either mistagged or not expressible in MiniZinc. In summary, we collect the top 36 tasks from Leetcode with the highest frequencies, 4 tasks in algorithmic contests, and 2 benchmarks from the previous approach.

Our benchmark consists of a wide range of classic dynamic programming tasks. Below we list some representatives. For COPs and CDPs, it consists of the Knapsack problem (KP), the longest increasing/common subsequence problem (LIS/LCS), the shortest path problem on grids, the maximum segment/independent sum problem, and the maximal multi-marketing problem. For CCPs, it consists of computing the Fibonacci/Catalan/binomial numbers and the counting variants of the KP/LCS/LIS.

***Baseline*.** We choose two baselines Sketch [Solar-Lezama et al. 2006] and FOSynth [Pu et al. 2011] as follows.

- Sketch is a general solver for sketch problems. We compare with Sketch since SynMem generates and completes the search sketch, and the comparison indicates the effectiveness of our synthesis procedure (Section 5.2). For each problem in our benchmark, we feed the generated search sketch (Figure 9) into Sketch.
- FOSynth is the state-of-the-art approach for synthesizing MAs from declarative specifications. FOSynth is also sketch-based, but their sketch template is less expressive than ours (Figure 9). The original implementation of FOSynth is unavailable. Hence, we acquired its reimplemented version by contacting an author of FOSynth. This implementation includes a built-in DSL and does not support easy change of DSL. Nevertheless, this DSL is a strict subset of the DSL used in our approach (Section 5.2).

***Procedure*.** We execute our implementation and the two baselines. We set the time limit as one hour for solving an individual benchmark. We obtain all results on the laptop with the Intel(R) Core(TM) i7-7820X CPU, 40GB RAM, and the Ubuntu 20.04 system.

***Results*.** Overall, SynMem solves 39/42 (92.8%) of our benchmarks. On 33 benchmarks, SynMem successfully finds the best algorithm. On the solved benchmarks, SynMem takes 2.01s on average. More specifically, on average, SynMem takes 1.91s on inductive synthesis and 0.10s on deductive term rewriting. Furthermore, after manually checking, we find that SynMem synthesizes the MPF with a minimal range on all solved benchmarks. Please refer to the full version of this paper for experimental results in detail, where we report the running time of SynMem, the time complexity

Table 1. The Comparison Result

	#Solved	#Failed (`runtime` > 1h)	#Best	AvgTime
Our approach	39	3	33	2.01s
Sketch	0	42	0	–
FoSynth	8	34	8	58.31s

of the synthesized MA (which is inspected manually), and the complexity of the reference answer for each benchmark.

The Comparison results are summarized in Table 1. For each approach, columns #Solved and #Failed are the number of solved and failed benchmarks, respectively. Column #Best is the number of benchmarks where the complexity of the synthesized program matches the complexity of the reference answer. Column AvgTime is the average running time per solved task.

Compared with the baseline, note that Sketch cannot synthesize any benchmark, and FOSynth solves 8 benchmarks with an average time of 58.31s and timeouts on 34 benchmarks. In contrast, SynMem successfully synthesizes 39 benchmarks in a shorter average time. Thus, in our benchmark, SynMem beats all baseline approaches.

***Discussion*.** Sketch fails on all benchmarks for the following reasons.

- First, the search sketch is too complex. It has about 30 lines and involves recursions, global array access, and modifications. In contrast, SynMem applies dedicated specifications for the holes based on the definition of LOF, MPF, and their updating functions.
- The dedicated specifications yield two independent tasks in each CEGIS iteration. SynMem solves them separately, leading to an efficient synthesis procedure. However, Sketch has to complete all holes in the search sketch simultaneously.

***Our limitations*.** SynMem fails on 3/40 tasks in our benchmark. This is because, on these benchmarks, the target memoization algorithm is out of reach of our template. Consider the following example of our failure from Leetcode 698:

> *Given an integer array* nums *and an integer* K, *return true if it is possible to divide this array into* K *non-empty subsets whose sums are all equal.*

We fail on this task since it requires us to synthesize the MPF that produces an unbounded list rather than a tuple. However, our method only supports the MPF that outputs tuples of a fixed dimension. Considering more forms of MPF is the future work.

On 9/37 benchmarks, SynMem synthesizes a sub-optimal MA that has a polynomial gap on the complexity with the reference answer. This is because these benchmarks require further algorithmic techniques other than memoization. Consider the following example from Leetcode 115:

> *Given two strings s and t ($|s| = n, |t| = m$), return the number of distinct subsequences of s which equals t.*

SynMem successfully synthesizes the basic $O(mn^2)$ MA. However, this problem requires an extra data structure for range query [He et al. 2011] to optimize the MA into running time $O(mn)$.

7 RELATED WORK

***Deriving Memoization Algorithms*.** There have been multiple trials to derive memoization algorithms, which could be categorized as manual and automated approaches.

First, there are manual or semi-automated approaches. Some of them [Bird and de Moor 1997; Bird and Gibbons 2020; de Moor 1995; Morihata et al. 2014; Mu 2008] propose a calculational framework so that the user can manually make a step-by-step derivation of memoization algorithms. Others of them [Acar et al. 2003; Giegerich et al. 2004; Liu and Stoller 1999; Pettorossi and Proietti 1996; Sauthoff et al. 2011] requires the user to specify complete memoization algorithms in some DSL, including the MPF and the local objective. These approaches verify whether it is correct. In contrast, our approach could automatically synthesize the memoization algorithm from a declarative specification, where the user only needs to provide a high-level description.

Next, there are automated approaches [Lin et al. 2021; Pu et al. 2011] that are closely related to ours. Below, we compare them with ours separately.

Lin et al. [2021]'s approach also considers specifications in MiniZinc style. Following a simple deductive procedure, their approach directly transforms the constraints and the objective function into a fold expression. It is limited as follows. First, their transformation rules support only a limited set of operators. For example, they could not handle the `forall` operator to perform element-wise operations over arrays. Next, their transformation succeeds only when there is no constraint between any two elements in inductive data structures. In our benchmark, they are applicable to only 6/42(14.2%) benchmarks. We do not compare with this approach in the evaluation part since the implementation is not avaliable.

The other approach [Pu et al. 2011] is purely inductive. Similar to SynMem, it also uses a sketch template to synthesize dynamic programming algorithms. It then applies an optimized version of Sketch to solve the synthesis task. However, it handles dynamic programming algorithms with a fixed number of scalar values for memoization. Thus, It is not able to handle classic CPs such as 0-1 knapsack, our running example in Section 2. By contrast, we consider a much more fruitful template. We introduce the MPF to represent an unbounded number of values for memoization. Under our framework, their template could be viewed as a subclass of ours where the MPF simply outputs 1 on all subproblems. To handle the more general synthesis problem, we apply a new algorithm mixing deductive and inductive synthesis methods. In our benchmark, 11/42(26.1%) of our benchmarks fall into their sketch template. In our experiment (Section 6), their approach solves 8/42(19.1%) of our benchmarks.

Finally, all previous approaches do not consider CCPs. We first address CCPs via our versatile sketch template and a dedicated approach to synthesizing the MPF.

Program Synthesis. Our approach relates to previous work in program synthesis as follows.

Recursive Program Synthesis. Many existing methods synthesize recursive programs [Farzan et al. 2022; Feser et al. 2015; Hu et al. 2021; Itzhaky et al. 2021; Kitzelmann and Schmid 2006; Kneuss et al. 2013; Knoth et al. 2019; Lubin et al. 2020; Polikarpova and Sergey 2019]. However, as far as we know, no approach could scale up to the synthesis of a complex memoization algorithm, which often involves tens of lines of code.

Sketching. SynMem follows a template of search sketches and proposes a dedicated method to synthesize all holes in the search sketch. However, the general solver Sketch for program sketching uses a constraint-based method, completely blind to the rich information in the specification. We have compared with Sketch in detail in Section 6.

Synthesizing specialized algorithms. Other approaches have been proposed to synthesize a specialized class of algorithms automatically [Farzan and Nicolet 2017, 2021; Morita et al. 2007; Smith and Albarghouthi 2016]. However, none is concerned with deriving efficient memoization algorithms.

At a more specific level, our approach shares some similarities with the previous work [Farzan and Nicolet 2017]. Both approaches apply the term rewriting techniques to complex relational program synthesis tasks, reducing a relational synthesis task into a conventional SyGuS task. However, instead of directly applying rewriting techniques (as in [Farzan and Nicolet 2017]), SynMem first applies a CEGIS procedure to instantiate (C1) and (D1) on concrete instances, removing operators that are difficult to be coped with in a rewriting system, such as the summation operator Σ. In this way, the design of the rewriting system becomes much simpler, and in many cases, we can guarantee the success of rewriting.

Relational Program Synthesis. The synthesis conditions in Section 4 yield two relational program synthesis tasks. However, our synthesis tasks involve too many unknown functions, thus is beyond the reach of the previous approach [Wang et al. 2018] in this field. To handle these tasks, SynMem applies the term rewriting method to bypass the synthesis of a large proportion of unknown functions and reduces the quantified relational synthesis task into a conventional SyGuS task.

8 CONCLUSION

This paper addresses the automated synthesis of correct and efficient memoization algorithms from the given declarative specification. We first make a novel reduction from synthesizing memoization algorithms to two smaller program synthesis tasks. However, the generated synthesis tasks are still too complex to be resolved by existing synthesizers. Thus, we propose a novel synthesis algorithm that combines the deductive and inductive methods to solve these tasks. Our approach successfully synthesizes 39/42 problems, outperforming the baselines.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable feedback on this paper and the anonymous OOPSLA Artifacts reviewers for their suggestions for the artifact. This work is supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, and the National Natural Science Foundation of China under Grant No. 62161146003.

A DETAILS OF THE SPECIFICATION LANGUAGE

In this part, we will systematically introduce our specification of combinatorial problems. This section is organized as follows. We will firstly introduce the syntax in detail, together with a remark about our syntactical restrictions of specifications for pragmatic consideration and discussions for possible extensions. Next, we introduce core concepts related to our specification. In the end, we formulate the synthesis problem.

Generally speaking, due to the fact that many application domains of combinatorial problems could be naturally modeled as constraint descriptions, in this paper, we will model combinatorial problems as a system of constraints four certain set of variables, and we adopt a declarative programming language for users to formulate a combinatorial problem in high level. The programming language is a simplified version of MiniZinc[Nethercote et al. 2007], which is a widely-used declarative language for modeling real-world problems. The syntax of our language is described in Figure 10 in detail. Note that we have presented an example of our specification in Figure 1. In general, since a wide range of combinatorial problems involve relations over lists, to make the specification fruitful and cover these problems, our language considers integers and arrays as basic components. Intuitively, the integers and integer identifiers encode certain properties of an individual object, and arrays encode a list (or set) of objects, and the expression in our language is connected by primitive operators or recursive operators, which corresponds to encode relations and computations over individual objects and a list of objects respectively. To present our language, we first overview the language with its four individual modules, and then we will discuss the types and expressions of this language in detail. Finally, we discuss our syntactical limitations.

To specify a combinatorial problem, one needs to provide four separate modules: inputs, variables, constraints over variables, and the objection. The intuition of the four parts is illustrated as follows:

- Firstly, the input part consists of a sequence of identifier declarations, annotated with its domain. The part consists of all ingredients needed to specify a concrete problem instance, we will use I to represent the set of all input identifiers.
- Next, the variable part is also made up of identifier declarations with domains. The part consists of all components for specifying a solution, we will use V to represent all variable identifiers, and use $\mathcal{R}$ to represent its corresponding domain.
- Then, the constraint part includes a set of boolean expressions. The part encodes the validity of a solution.
- Finally, the objection part specifies the goal of this problem. Our language supports objections including, finding the best valid solution, finding any solution, or counting the number of solutions.

Type. In our specification, we consider ranges and arrays as basic types. A range R, formed by two basic integer expressions $BE_1 \,..\, BE_2$, corresponds to the bounded interval $[BE_1, BE_2]$ of integers, we also offer a syntactic sugar `int` that represents the value of all integers $[-\infty, +\infty]$. In our language, we explicitly separate variables and input, a range R with annotation `var` corresponds to the type signature for a variable, and a range R without this annotation corresponds to an input's signature. An array is specified by two ranges, where the first range represents the set of its indices, and the second range represents the domain of the array's elements.

Expression. Our specification consists of boolean expressions and integer expressions. Intuitively, boolean expressions offer users to encode constraints, and integer expressions are used to encode the arithmetics. Boolean expressions could have three forms as follows:

- Standard boolean constant `true` and `false`.

- Expressions connected by primitive boolean operator $PrimOp^{\text{bool}}$. We consider basic widely-used primitive operators such as $\wedge, \vee, \geq, \leq, <, >, =$ and $\neq$. (e.g., $x < 3$, $(x \neq 3) \wedge (y > 5)$).
- Expressions connected by recursive operator **forall**, which is intuitively used to set up constraints over list. The **forall** expression is constructed by the iterating variable *id*, the iteration range *R*, and the iteration body E^{bool}. It will recursively iterate the variable *id* from its range *R*, and specify the constraint formed by instantiating the variable *id*. For example, if we want to encode the constraint that an array `a` with index `1..n` is monotonically increasing, we could specify that `forall(i in 1..n-1)(a[i]<a[i+1]);`

Similarly, an integer expression could also have three forms as follows:

- Constant value *val* and identifier *id* (e.g., $3, x$).
- Expressions connected by primitive integer operator $PrimOp^{\text{int}}$, where consider common operators such as $+, -, \times, /$, max and min. (e.g., $x \times 3$, $\max\{x, y\}$).
- Expressions connected by recursive operator, where we consider **sum**, **product**, **max**, **min**. These operators are intuitively used to accumulate a primitive operator over a list. Similar to **forall**, **sum** (**product**, **max**, **min**, respectively) will recursively caluculates the summation (production, maximum, minimum, respectively) of each body term E^{int} formed by instantiating the variable *id*. For example, if we want to compute the square sum of an array `a` with index `1..n`, we could specify that `sum(i in 1..n)(a[i]×a[i]);`

Objection operator *objOp*. Our language offers four types of objections. If we are interested in finding the best valid solution, we could write `solve maximize` or `solve minimize`, together with the objective function to be maximized (or minimized). If we only want to find an arbitrary valid solution or count the number of solutions, we could write `solve satisfy`, or `count satisfy` respectively.

Syntatic Restrictions. Though our language has supported various widely-used primitive and recursive operators, we do not support the recursive operator `exists` that checks one of a set of constraints is satisfied. As a result, our specification restricts combinatorial problems that could be formulated by first-order $\exists\forall$ formulas. However, this type of formula forms a fruitful and widely-studied subclass of first-order logic. Moreover, as presented in Section 6, our language admits a large number of different combinatorial problems. It is left as future work to extend our algorithm to support more primitive and recursive operators, and support a wider class of formulas.

Specification	$Spec$	$\rightarrow I^*V^*C^*O$
Input	I	$\rightarrow T^I : id$
InputType	T^I	$\rightarrow R \mid \texttt{array[}R\texttt{] of } R$
Variable	V	$\rightarrow T^v : varid$
Constraint	C	$\rightarrow \texttt{constraint } E^{\texttt{bool}}$
Objective	O	$\rightarrow objOp \mid objOp\ E^{\texttt{varint}}$
VarType	T^v	$\rightarrow \texttt{var } R \mid \texttt{array[}R\texttt{] of var } R$
Range	R	$\rightarrow \texttt{int} \mid BE^{\texttt{int}}\ \texttt{..}\ BE^{\texttt{int}}$
VarRange	$R^{\texttt{var}}$	$\rightarrow \texttt{int} \mid BE^{\texttt{varint}}\ \texttt{..}\ BE^{\texttt{varint}}$
Expression	$E^{\texttt{var}}$	$\rightarrow E^{\texttt{bool}} \mid E^{\texttt{int}} \mid E^{\texttt{varint}}$
BoolExp	$E^{\texttt{bool}}$	$\rightarrow \texttt{true} \mid \texttt{false} \mid PrimOp^{\texttt{bool}}(E_1^{\texttt{var}}, \cdots, E_k^{\texttt{var}}) \mid \texttt{forall(}id\ \texttt{in}\ R\texttt{)(}E^{\texttt{bool}}\texttt{)}$
IntExp	$E^{\texttt{int}}$	$\rightarrow BE^{\texttt{int}} \mid RE^{\texttt{int}}$
BaseIntExp	$BE^{\texttt{int}}$	$\rightarrow val \mid id \mid arrayid[BE^{\texttt{int}}] \mid PrimOp^{\texttt{int}}(E_1^{\texttt{int}}, \cdots, E_k^{\texttt{int}})$
RecursiveIntExp	$RE^{\texttt{int}}$	$\rightarrow RecursiveOp^{\texttt{int}}(rid\ \texttt{in}\ R^{\texttt{var}})(E^{\texttt{int}})$
VarIntExp	$E^{\texttt{varint}}$	$\rightarrow BE^{\texttt{varint}} \mid RE^{\texttt{varint}}$
BaseVarIntExp	$BE^{\texttt{varint}}$	$\rightarrow BE^{\texttt{int}} \mid varid \mid vararrayid[BE^{\texttt{varint}}] \mid PrimOp^{\texttt{int}}(E_1^{\texttt{var}}, \cdots, E_k^{\texttt{var}})$
RecursiveVarIntExp	$RE^{\texttt{varint}}$	$\rightarrow RecursiveOp^{\texttt{int}}(rid\ \texttt{in}\ R^{\texttt{var}})(E^{\texttt{varint}})$

Fig. 10. The syntax of our specificaiton

B PROOF

Proof for proposition 5.2. Assume that each $p \in \mathcal{L}$ covers $< \frac{1}{\mathsf{K}}$ fraction of the conditions, then for all m programs $p_1, \cdots, p_m \in \mathcal{L}(m \leq \mathsf{K})$, they can cover $< \frac{m}{\mathsf{K}} \leq 1$ fraction of the conditions at most, which contradicts the fact that exists $\leq \mathsf{K}$ programs can cover all conditions. □

sectionDetails of Generating Search algorithms

Given the enumeration order, there is a trivial method that syntactically translates a logic specification to a recursive function that can compute its solution. The basic idea is to enumerate the variables in a specific order while maintaining a set C_{now} that contains constraints involving the variables that have not been enumerated. That is, the truth values of constraints in C_now are unable to be determined yet. Here is the framework of the function:

- Initially, C_{now} is set to C in the specification.
- Enumerate the variables according to the given order. For each variable, we should
  - Choose a possible value in the variable's domain.
  - Remove the constraints that can be determined from C_{now} and update it.
  - Go into the enumeration for the next variable recursively.
  - Choose the next possible value and repeat the above steps.
- After all variables are enumerated, we get a complete assignment, and C_{now} should be an empty set now. Finally, we update the current optimal value.

After the translation, we expect to get an enumerative search algorithm in figure 4. We can roughly divide the translation process into two parts: enumeration and computation. We will illustrate these two parts in detail and answer the following questions later:

- How do we model the enumeration order?
- How do we compute the satisfiability of a constraint and the value of the objective function?
- How do we maintain C_{now}?

Algorithm 4: Enumerative Search Algorithm

```
Function Search($i$, $A$, $C_{now}$):
    if $i > |V|$ then
        opt := max(opt, EvalOptimum (O));
    else
        foreach $\alpha \in R(\tilde{v}_{i+1})$ do
            $C'_{now}$ := Update($\tilde{v}_{i+1}$,$\alpha$,$C_{now}$);
            $A'$ := $A \cup \{\tilde{v}_{i+1} : \alpha\}$;
            sat := true;
            foreach $c \in C'_{now}$ do
                if Determined($c$, $A'$) then
                    if EvalConstraint($c$) = $true$ then Remove($C_{now}$,c);
                    else sat := false;
                end
            end
            if $sat = true$ then Search($i$+1, $A$, $C'_{now}$);
        end
    end
```

Enumeration order. A enumeration order is an order on V, the set of variables. The order produces a list $\tilde{V}$ that contains all variables in V. For example, if a specification only have two integer variable a, b and a array variable $c[l..r]$ (i.e., $V = \{a, b, c_l \cdots, c_r\}$), then both $\tilde{v} = [c_l, \cdots, c_r, b, a]$ and $\tilde{v} = [a, c_r \cdots c_l, b]$ can be valid orders. Note that the range of indices of c, i.e., the values of l and r, are determined once the input I is given.

After we have enumerated a part of the variables, we need to define a state that records the current partial assignment and the rest of the variables. We model the state as a tuple (i, A), which means we have already determined the value of variables $\tilde{v}_1, \cdots, \tilde{v}_i$, and A is a dictionary that records assignment. In the previous example, assume $l = 2, r = 4, \tilde{v} = [a, c_2, c_3, c_4, b]$, and we have enumerated $a = 4, c_2 = 1$, we will get $i = 2$ and $A = \{a : 4, c_2 : 1\}$.

For each recursive call to the search function, we first take out $\tilde{v}_{i+1}$, which is the variable currently enumerated. After we choose a value α from $\tilde{v}_{i+1}$'s domain, we add $\tilde{v}_{i+1} : \alpha$ to the assignment A. Then we update C_{now} and go to the next variable by recursively calling the search function with index $i + 1$ and new assignment A.

The way we model the enumeration order is natural and is close to the normal style in which programmers write an enumerative search algorithm. Moreover, *assn* and *rest* are important since our synthesis of the memoization algorithm is based on them.

Computation Part. The computation for the value of a constraint or the objective function is purely syntactical. Given a assignment A to variables, the functions `EvalConstraint`(c, A) and `EvalOptimum`(c, A) in algorithm 4 implement the evaluation of a constraint and the objective function respectively. The semantics of these functions are in figure 11.

$E \Downarrow_e^A P$ denotes that the expression E is able to be evaluated to value P under the assignment A. The rule E_RecursiveOp is important, which translates the operator to a fold-style recursive function. For example, if the recursive operator is *sum*, then the *InitValue* and *CombineOp* are 0 and + respectively.

E_PrimOp
$$\frac{E_i \Downarrow_e^A P_i (i = 1, \cdots, k)}{PrimOp(E_1, \cdots, E_k) \Downarrow_e^A PrimOp(P_1, \cdots, P_k)}$$

E_Range
$$\frac{R \longrightarrow E_1..E_2 \qquad E_1 \Downarrow_e^A P_1 \qquad E_2 \Downarrow_e^A P_2}{R \Downarrow_e^A P_1..P_2}$$

E_RecursiveOp
$$\frac{R \Downarrow_e^A l..r \qquad E \Downarrow_e^A P \qquad InitValue(RecursiveOp) = c \qquad CombineOp(RecursiveOp) = \circ}{RecursiveOp(rid \texttt{ in } R)(E) \Downarrow_e \texttt{let fun } f(i) = (i > r ? c : ([i/id]P) \circ f(i+1)) \texttt{ in } f(l)}$$

E_VarId
$$\frac{}{varid \Downarrow_e^A get(A, varid)}$$

E_Id
$$\frac{}{id \Downarrow_e^A get(I, id)}$$

E_VarArray
$$\frac{E \Downarrow_e^A P}{vararrayid[E] \Downarrow_e^A get(A, vararrayid[P])}$$

E_Array
$$\frac{E \Downarrow_e^A P}{arrayid[E] \Downarrow_e^A get(I, id[P])}$$

E_Const
$$\frac{}{val \Downarrow_e^A val}$$

Fig. 11. The rules of `EvalConstraint()` and `EvalOptimum()`

Maintaining C_{now}. The set C_{now} contains constraints whose satisfiability cannot yet be confirmed. The purpose of maintaining this set is to prune out unnecessary enumerations: If a constraint can be determined to be false under a partial assignment, the enumerated algorithm will try the next possible value of the current variable, rather than go into the next variable. The enumerative search algorithm is based on the following two ideas to maintain C_{now}:

1. For a constraint in C_{now}, it may only be relevant to a subset of variables. If all variables in this subset have been determined, the truth value of the constraint can be computed.
   - If it is true, it will be removed from C_{now}.
   - Otherwise, the algorithm will stop considering the next variable.
2. For a constraint with form `forall`$(id$ `in` $R)(E^{\texttt{bool}})$, if all variables in R are determined, then the constraint can be broken into $|R|$ subconstraints.

For idea 1, the function `Determined()` in algorithm 4 implements it. `Determined(c,assn)` means the satisfibility of constraint c can be determined provided the partial assignment A. The detailed semantics of the function are in figure 12. $E \prec_c A$ denotes that the value of expression E can be determined only given the assignment A. Note the rules C_RangeDetermined and C_RangeUndetermined, both of whom are about RecursiveOp. Intuitively, The First one says that if the lower and upper bounds of the range R can be evaluated to l and r, then the algorithm adds an item $rid : l..r$ to Ω. *Omega* is a map to record the recursive identifiers' ranges that are determined, which might be useful to determine which items in a var array are relevant to the constraint later. To be specific, when we meet a $vararrayid[E]$ and all of the recursive identifiers occurred in E have a determined range, we are able to compute the set of items that is relevant. On the other hand, if some of the recursive identifiers are undetermined, we consider all items of the array relevant to constraint. The rules C_VarArrayRidCompleted and C_VarArrayRidIncompleted describe these features formally.

C_PrimOp

$$\frac{\Omega; E_i \prec_c A_i (i = 1, \cdots, k)}{\Omega; PrimOp(E_1, \cdots, E_k) \prec_c \bigcup_{i=1}^{k} A_i}$$

C_RangeDetermined

$$\frac{\Omega; R \prec_c A_R \qquad R \Downarrow_e^{A_R} l..r \qquad (\Omega, id : l..r); E \prec_c A_E}{\Omega; RecursiveOp(rid \ \texttt{in} \ R)(E) \prec_c A_R \cup A_E}$$

C_RangeUndetermined

$$\frac{\Omega; R \prec_c A_R \qquad \Omega; E \prec_c A_E}{\Omega; RecursiveOp(rid \ \texttt{in} \ R)(E) \prec_c A_R \cup A_E}$$

C_VarId

$$\frac{}{\Omega; varid \prec_c \{varid\}}$$

C_Id

$$\frac{}{\Omega; id \prec_c \varnothing}$$

C_ArrayId

$$\frac{\Omega; E \prec_c A}{\Omega; arrayid[E] \prec_c A}$$

C_VarArrayRidCompleted

$$\frac{AllOfRid(E) \subseteq \Omega \qquad K = \{x \mid E \Downarrow_e^{A \cup A_\Omega} x, \forall A_\Omega \text{ is valid under } \Omega\}}{\Omega; vararrayid[E] \prec_c A \cup \{vararrayid[a] \mid a \in K\}}$$

C_VarArrayRidIncompleted

$$\frac{AllOfRid(E) \subsetneq \Omega}{\Omega; vararrayid[E] \prec_c A \cup \{vararrayid[a] \mid a \in \text{index range}\}}$$

Fig. 12. The rules of `Determined()`

Therefore, the enumerative search algorithm can advance the computation of a constraint as soon as all relevant variables have been assigned.

For idea 2, the algorithm decomposes `forall`-form constraints when the range is determined, which is done during the `Update()` procedure in algorithm 4. The updating rule is described in figure 13.

$$\textsc{Decompose}\quad \frac{R \prec_c A \qquad R \Downarrow_e^A l..r \qquad \langle \texttt{forall}(id \ \texttt{in} \ R)(E^{\text{bool}})\rangle \in C_{now}}{\forall i \in l..r, [i/id]E \in C'_{now}}$$

Fig. 13. The rule of decomposing `forall`-form constraints

C DETAILS OF OUR SYNTHESIS RESULT

REFERENCES

[n. d.]. Full version of this paper. https://boyvolcano.github.io/publication/oopsla-23/oopsla23.pdf

[n. d.]. National Olympiad in Informatics in Provinces-Junior. https://noi.ccf.org.cn/zxzy/lnzl/index.shtml

[n. d.]. The world's leading online programming learning platform. https://leetcode.com/

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2003. Selective Memoization. *SIGPLAN Not.* 38, 1 (jan 2003), 14–25. https://doi.org/10.1145/640128.604133

Heimo H. Adelsberger. 2003. Prolog Programming Language. In *Encyclopedia of Physical Science and Technology (Third Edition)* (third edition ed.), Robert A. Meyers (Ed.). Academic Press, New York, 155–178. https://doi.org/10.1016/B0-12-227410-5/00853-X

Alfred V. Aho and John E. Hopcroft. 1974. *The Design and Analysis of Computer Algorithms* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

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

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 13–24. https://doi.org/10.1145/3368089.3409757

Roman Barták. 1999. Constraint programming: In pursuit of the holy grail. *Proceedings of WDS99 (invited lecture)* (01 1999).

Richard Bird and Oege de Moor. 1997. *Algebra of Programming.* Prentice-Hall, Inc., USA.

Richard Bird and Jeremy Gibbons. 2020. *Algorithm Design with Haskell.* Cambridge University Press. https://doi.org/10.1017/9781108869041

Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. 2022. *An Efficient Branch-and-Bound Solver for Hitting Set.* 209–220. https://doi.org/10.1137/1.9781611977042.17 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611977042.17

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 427–442.

Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. 2011. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning* 47, 4 (2011), 341–367.

Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1027–1040. https://doi.org/10.1145/3314221.3314596

Norman H. Cohen. 1983. Eliminating Redundant Recursive Calls. *ACM Trans. Program. Lang. Syst.* 5, 3 (jul 1983), 265–299. https://doi.org/10.1145/2166.2167

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Oege de Moor. 1995. A generic program for sequential decision processes. In *Programming Languages: Implementations, Logics and Programs*, Manuel Hermenegildo and S. Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23.

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA,

Table 2. The Details of Our Synthesis Result

Benchmark	Runtime (s)	Deductive Time(s)	Inductive Time(s)	Complexity	Best	Match Best?
1049	fail	-	-	-	$O(n^3)$	✗
1143	4.128	4.045	0.083	$O(n^6)$	$O(n^2)$	✗
115	0.248	0.234	0.015	$O(mn^2)$	$O(mn)$	✗
118	0.481	0.454	0.027	$O(n^2)$	$O(n^2)$	✓
120	0.699	0.659	0.041	$O(n^3)$	$O(n^3)$	✓
121	6.205	5.937	0.267	$O(n)$	$O(n)$	✓
122	0.72	0.679	0.041	$O(n)$	$O(n)$	✓
123	4.107	4.03	0.076	$O(n)$	$O(n)$	✓
152	fail	-	-	-	$O(n)$	✗
198	0.902	0.86	0.042	$O(n)$	$O(n)$	✓
213	2.799	2.576	0.223	$O(n)$	$O(n)$	✓
22	0.535	0.494	0.041	$O(n^2)$	$O(n^2)$	✓
279	0.564	0.538	0.026	$O(n^3)$	$O(n^3)$	✓
300	0.298	0.284	0.015	$O(n^4)$	$O(n^2)$	✗
309	2.582	2.363	0.22	$O(n)$	$O(n)$	✓
32	8.643	8.466	0.178	$O(n^2)$	$O(n)$	✓
322	0.590	0.563	0.027	$O(nC)$	$O(nC)$	✓
354	0.117	0.102	0.014	$O(n^4)$	$O(n^2)$	✗
392	0.316	0.301	0.015	$O(n^3)$	$O(n^3)$	✓
416	0.528	0.501	0.027	$O(nS)$	$O(nS)$	✓
45	2.286	2.089	0.197	$O(n^4)$	$O(n^2)$	✗
474	3.936	3.858	0.077	$O(nC_1C_2)$	$O(nC_1C_2)$	✓
494	0.538	0.511	0.027	$O(nS)$	$O(nS)$	✓
509	2.200	2.011	0.188	$O(n)$	$O(n)$	✓
516	4.276	4.187	0.089	$O(n^6)$	$O(n^2)$	✓
53	4.949	4.818	0.130	$O(n)$	$O(n)$	✓
55	0.311	0.297	0.014	$O(n)$	$O(n)$	✓
62	0.584	0.558	0.027	$O(n^2)$	$O(n^2)$	✓
646	0.153	0.139	0.014	$O(n^4)$	$O(n^2)$	✓
673	0.210	0.196	0.014	$O(n^4)$	$O(n^2)$	✓
698	fail	-	-	-	$O(2^k)$	✗
70	2.144	1.962	0.182	$O(n)$	$O(n)$	✓
714	6.444	6.171	0.273	$O(n)$	$O(n)$	✓
746	2.550	2.337	0.214	$O(n)$	$O(n)$	✓
873	2.797	2.574	0.223	$O(n^5)$	$O(n^2)$	✗
96	0.841	0.801	0.040	$O(n^2)$	$O(n^2)$	✓
Comp1	0.556	0.530	0.026	$O(nB)$	$O(nB)$	✓
Comp2	0.448	0.422	0.027	$O(nS)$	$O(nS)$	✓
Comp3	0.552	0.526	0.026	$O(nmS)$	$O(nmS)$	✓
Comp4	0.160	0.146	0.014	$O(n^3)$	$O(n^3)$	✓
assem	3.053	2.816	0.236	$O(n)$	$O(n)$	✓
mas	4.953	4.827	0.126	$O(n)$	$O(n)$	✓

USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/3519939.3523726

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of Divide and Conquer Parallelism for Loops. *SIGPLAN Not.* 52, 6 (jun 2017), 540–555. https://doi.org/10.1145/3140587.3062355

Azadeh Farzan and Victor Nicolet. 2021. Phased Synthesis of Divide and Conquer Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 974–986. https://doi.org/10.1145/3453483.3454089

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *SIGPLAN Not.* 50, 6 (jun 2015), 229–239. https://doi.org/10.1145/2813885.2737977

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

Meng He, J. Ian Munro, and Patrick K. Nicholson. 2011. Dynamic Range Selection in Linear Space. In *Algorithms and Computation*, Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 160–169.

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2021. Synthesis with Asymptotic Resource Bounds. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 783–807. https://doi.org/10.1007/978-3-030-81685-8_37

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages* 5 (2021), 1 – 28.

Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems.* Springer. I–XX, 1–546 pages.

Emanuel Kitzelmann and Ute Schmid. 2006. Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. *J. Mach. Learn. Res.* 7 (dec 2006), 429–454.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. https://doi.org/10.1145/3314221.3314602

Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-Recursive Expressions. *Proc. ACM Program. Lang.* 7, POPL, Article 70 (jan 2023), 31 pages. https://doi.org/10.1145/3571263

Shu Lin, Na Meng, and Wenxin Li. 2021. *Generating Efficient Solvers from Constraint Models.* Association for Computing Machinery, New York, NY, USA, 956–967. https://doi.org/10.1145/3468264.3468566

Yanhong A. Liu and Scott D. Stoller. 1999. Dynamic Programming via Static Incrementalization. In *Programming Languages and Systems*, S. Doaitse Swierstra (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–305.

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (aug 2020), 29 pages. https://doi.org/10.1145/3408991

David Maier. 1978. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM* 25, 2 (apr 1978), 322–336. https://doi.org/10.1145/322063.322075

Claude Marché. 1996. Normalized Rewriting: An Alternative to Rewriting modulo a Set of Equations. *J. Symb. Comput.* 21, 3 (mar 1996), 253–288. https://doi.org/10.1006/jsco.1996.0011

Claude Marché and Xavier Urbain. 1998. Termination of Associative-Commutative Rewriting by Dependency Pairs. In *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA '98)*. Springer-Verlag, Berlin, Heidelberg, 241–255.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Akimasa Morihata, Masato Koishi, and Atsushi Ohori. 2014. Dynamic Programming via Thinning and Incrementalization. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 186–202.

Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. *SIGPLAN Not.* 42, 6 (jun 2007), 146–155. https://doi.org/10.1145/1273442.1250752

Shin-Cheng Mu. 2008. Maximum Segment Sum is Back Deriving Algorithms for Two Segment Problems with Bounded Lengths. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 31–39. https://doi.org/10.1145/1328408.1328414

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming – CP 2007*, Christian Bessière (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 529–543.

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

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (jan 2019), 30 pages. https://doi.org/10.1145/3290385

M. Presburger. 1931. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt.* https://books.google.com.sg/books?id=7agKHQAACAAJ

Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. 2011. Synthesis of First-Order Dynamic Programming Algorithms. *SIGPLAN Not.* 46, 10 (oct 2011), 83–98. https://doi.org/10.1145/2076021.2048076

Georg Sauthoff, Stefan Janssen, and Robert Giegerich. 2011. Bellman's GAP - A Declarative Language for Dynamic Programming. *PPDP'11 - Proceedings of the 2011 Symposium on Principles and Practices of Declarative Programming*, 29–40. https://doi.org/10.1145/2003476.2003484

C. Schensted. 1961. Longest Increasing and Decreasing Subsequences. *Canadian Journal of Mathematics* 13 (1961), 179–191. https://doi.org/10.4153/CJM-1961-015-3

Alexander Schrijver. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Vol. B.

Robert E. Shostak. 1979. A Practical Decision Procedure for Arithmetic with Function Symbols. *J. ACM* 26, 2 (apr 1979), 351–360. https://doi.org/10.1145/322123.322137

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. *SIGPLAN Not.* 51, 6 (jun 2016), 326–340. https://doi.org/10.1145/2980983.2908102

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (oct 2006), 404–415. https://doi.org/10.1145/1168919.1168907

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (oct 2018), 27 pages. https://doi.org/10.1145/3276525

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

Received 2023-04-14; accepted 2023-08-27