Tunneling Through the Hill: Multi-Way Intersection for Version-Space Algebras in Program Synthesis

GUANLIN CHEN[*], Peking University, China
RUYI JI[*], Peking University, China
SHUHAO ZHANG[*], Peking University, China
YINGFEI XIONG[†], Peking University, China

Version space algebra (VSA) is an effective data structure for representing sets of programs and has been extensively used in program synthesis. Despite this success, a crucial shortcoming of VSA-based synthesis is its inefficiency when processing many examples. Given a set of IO examples, a typical VSA-based synthesizer runs by first constructing an individual VSA for each example and then iteratively intersecting these VSAs one by one. However, the intersection of two VSAs can be much larger than the original ones – this effect accumulates during the iteration, making the scale of intermediate VSAs quickly explode.

In this paper, we aim to reduce the cost of intersecting VSAs in synthesis. We investigate the process of the iterative intersection and observe that, although this process may construct some huge intermediate VSAs, its final VSA is usually small in practice because only a few programs can pass all examples. Utilizing this observation, we propose the approach of *multi-way intersection*, which directly intersects multiple small VSAs into the final result, thus avoiding the previous bottleneck of constructing huge intermediate VSAs. Furthermore, since the previous intersection algorithm is inefficient for multiple VSAs, we design a novel algorithm to avoid most unnecessary VSA nodes.

We integrated our approach into two SOTA VSA-based synthesizers: a general synthesizer based on VSA and a specialized one for the string domain Blaze. We evaluate them over 4 different datasets, 994 synthesis tasks; the results show that our approach can significantly improve the performance of VSA-based synthesis, with up to 105 more tasks solved and a speedup of 7.36×.

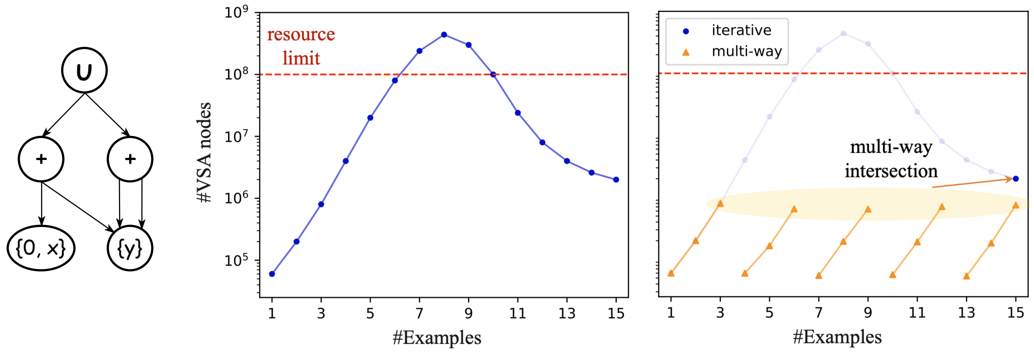
1 INTRODUCTION

Programming by example (PBE) [Shaw et al. 1975] is an important paradigm of program synthesis, where the goal is to learn a program from a set of input-output (IO) examples. PBE has attracted much research interest because many practical synthesis tasks can be converted into PBE through the OGIS framework [Jha and Seshia 2017]. After decades of development, there have been various effective PBE solvers proposed for different domains [Alur et al. 2015; Ding and Qiu 2024; Dong et al. 2022; Gulwani 2011; Ji et al. 2021; Lee 2021; Lee et al. 2018; Li et al. 2024; Mitchell 1982; Padhi et al. 2018; Reynolds et al. 2019; Udupa et al. 2013; Wang et al. 2017]. Among these solvers, a particular branch based on *version spaces* [Mitchell 1982] has recently achieved great success in various scenarios, including string manipulation [Gulwani 2011], web automation [Dong et al.

[*]Equal contribution to this paper
[†]Corresponding author

Authors' addresses: Guanlin Chen, Peking University, Beijing, China, cgl@stu.pku.edu.cn; Ruyi Ji, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Shuhao Zhang, Peking University, Beijing, China, shuhaozhang25@stu.pku.edu.cn; Yingfei Xiong, Peking University, Beijing, China, xiongyf@pku.edu.cn.

ACM XXXX-XXXX/2025/8-ART
https://doi.org/10.1145/3763176



(a) A sample VSA. (b) Iterative Intersection. (c) Our multi-way intersection.

Fig. 1. (a) A sample VSA that represents $\{0 + y, x + y, y + y\}$. (b, c) The scale of the VSA during the intersection, shown on a logarithmic scale. The red line marks a resource limit at VSAs with 10^8 nodes. Such a VSA typically takes several minutes to construct and tens of GBs of memory to store, unacceptable in practice.

2022; Li et al. 2024], and data processing [Padhi et al. 2018; Wang et al. 2017]. Its key idea is to construct a sub-space of programs (i.e., the version space) satisfying the given IO examples and then extract the best synthesis result from the space, typically via a pre-defined ranking function.

VSA-based synthesis. *Version space algebra (VSA)* is an effective data structure for learning version spaces. It follows a graph-like structure, as shown in Fig. 1a, where each node represents a set of programs, and the edges specify how programs are combined from small to large. Its advantage is to compactly represent a large set of programs, usually with an exponential saving on memory; so a typical VSA-based synthesizer will construct a VSA for the set of valid programs satisfying all examples, then efficiently extract the best result through the compact structure of this VSA.

Despite the success, one major issue of existing VSA-based synthesizers is their inefficiency when processing many examples. This issue originates from the iterative process of intersecting VSAs; given a set of IO examples, the existing synthesizers will first construct a series of single-example VSAs, each representing the set of programs satisfying one example, and then intersect these VSAs one by one. However, the intersection of two VSAs can be much larger than the original ones — in the worst case, its size can be the product of the original sizes [Polozov and Gulwani 2015]. This effect can accumulate during the iteration, making the VSA scale quickly exceed the resource limit.

Multi-way intersection. This paper aims to accelerating VSA-based synthesis by *reducing the cost of intersecting VSAs.* Our approach is motivated by the following observation.

- **Observation**. The size of the intersected VSA typically exposes a hill-like curve during the iteration, as shown in Fig. 1b. Although the size may start with a quick explosion, it will switch to going down after reaching a peak and finally fall to a fairly low level; during this process, the size of the largest intermediate VSA can be orders of magnitude larger than the initial and final ones — approximately 300× on average in our dataset.

This property stems from the fact that practical synthesis systems are typically eager for *disambiguation* [Le et al. 2017; Solar-Lezama et al. 2006]. For such systems, their primary mission is always to find a program that meets the user's demand. To ensure this, they need to collect enough information (typically examples) to exclude most invalid programs; then, a small VSA will be enough to represent the few remaining programs.

Based on this observation, we propose a novel approach to intersect VSAs, *multi-way intersection*, to avoid generating the giant VSAs at the peak. As shown in Fig. 1c, our approach divides the examples into several groups, iteratively constructs a VSA for each group, and then intersects these VSAs at once. In this way, it excavates a tunnel through the hill, thus skipping the previous bottleneck at the peak.

To make our approach effective, the key is to find an efficient algorithm for intersecting many VSAs at once. However, existing algorithms are far from efficient enough. For example, a naive algorithm constructs the intersected VSA by ranging over node tuples $(n_1, \dots, n_k)$ where n_i comes from the ith input VSA. For each tuple, this algorithm constructs a node for the intersected VSA, representing the set of programs shared among the nodes in the tuple, and adds edges correspondingly. This algorithm cannot support an efficient multi-way intersection because it enumerates all node tuples, which requires exponential time and can be even slower than the iterative intersection. Although Polozov and Gulwani [2015] have proposed a better algorithm for VSA intersection, we prove it still suffers from a similar issue when the number of examples increases (Thm. 3.7).

Our intersection algorithm. To address this issue, we utilize again our observation that the final VSA is typically small. In this case, the naive algorithm will waste most of its time on constructing empty nodes that represent no valid program. Such nodes are trivially unnecessary, so skipping them from the enumeration could provide a significant speed-up.

Following this idea, our algorithm builds the intersection from the bottom up and uses lower non-empty nodes to precisely construct the higher non-empty ones. It is powered by a search method based on the *trie* data structure [Briandais 1959] to efficiently locate non-empty nodes, thus implicitly ignoring empty ones. We demonstrate the advantages of this algorithm in theory.

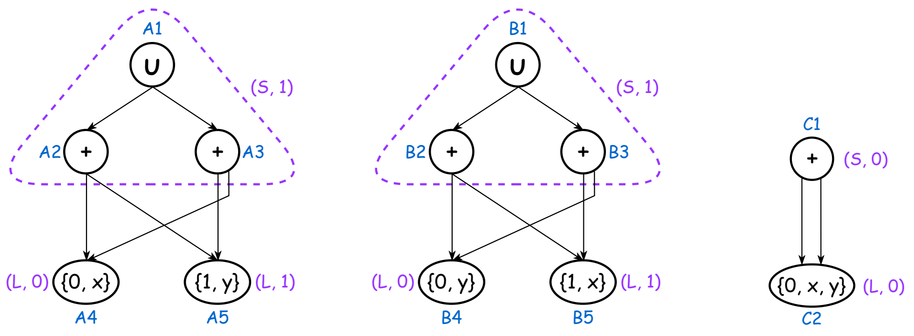
- On a case study over commutative and associative operators, we prove our algorithm can avoid an exponential explosion observed in the previous best algorithm for VSA intersection [Polozov and Gulwani 2015].
- By integrating our algorithm into a VSA-based synthesizer, we prove its efficiency is no worse than *observational equivalence* [Udupa et al. 2013], a SOTA general synthesizer based on enumeration, even in the worst case.

Evaluation. We implement our approach in a tool for intersecting VSAs, named Mole, since it tunnels through the hill of intermediate VSAs (Fig. 1c) just like moles. For evaluation, we integrated Mole into two SOTA VSA-based synthesizers, a general synthesizer based on finite tree automata (FTA)[1] [Wang et al. 2017] and a specialized synthesizer Blaze relying on abstract refinement [Wang et al. 2018]. We evaluate both synthesizers over 4 different datasets, 994 synthesis tasks; the results show that our approach can significantly improve the performance of VSA-based synthesizers, with up to 105 more tasks solved and a speedup of 7.36×.

Contributions. To sum up, this paper makes the following contributions.

- Defining the multi-way intersection problem and analyzing the inefficiency of the existing intersection algorithm (Sec. 3).
- Proposing an efficient bottom-up algorithm for intersecting multiple VSAs (Sec. 4).
- Implementing our approach into two synthesizers and evaluating its performance over 4 datasets and 994 synthesis tasks (Sec. 5).

[1]Strictly speaking, FTA is a different data structure for learning version spaces. However, as we shall mention later, this paper considers only finite program spaces, where FTA is proven equivalent to VSA [Koppel 2021]. Therefore, we shall use FTA and VSA interchangeably for simplicity.



(a) the VSA for example e_a. (b) the VSA for example e_b. (c) the VSA for example e_c.

Fig. 2. The single-example VSAs in our sample task, where the topmost node is the root, and e_a, e_b, e_c denotes $\langle x = 0, y = 1\rangle \mapsto 1$, $\langle x = 1, y = 0\rangle \mapsto 1$, and $\langle x = 0, y = 0\rangle \mapsto 0$, respectively. Here, we use blue to assign an index for each VSA node, and use violet to mark the corresponding non-terminal and the output value.

2 OVERVIEW

This section gives an overview of our approach with a PBE task. Its goal is to synthesize the program $x + y$ (or its equivalent form $y + x$) from the following grammar.

$$S \coloneqq L + L \qquad L \coloneqq 0 \mid 1 \mid x \mid y$$

This task provides three IO examples for synthesis, as listed below.

$$(e_a)\ \langle x = 0, y = 1\rangle \mapsto 1 \qquad (e_b)\ \langle x = 1, y = 0\rangle \mapsto 1 \qquad (e_c)\ \langle x = 0, y = 0\rangle \mapsto 0$$

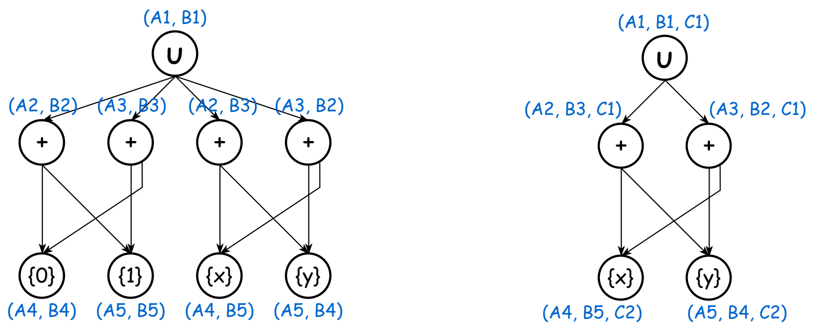
2.1 Background: VSA and VSA-Based Synthesis

A VSA can be viewed as a directed graph with a root node, as illustrated in Fig. 2a. Each VSA node represents a set of programs, and the whole VSA represents the program set of its root. There are three kinds of VSA nodes, corresponding to three different ways to construct programs.

- Each *leaf* node is labeled with the set of programs it represents, such as nodes A_4 and A_5.
- Each *join* node is labeled with an operator, such as nodes A_2 and A_3. It constructs programs by applying the labeled operator to the programs represented by its children. For example, A_2 applies operator + by picking the left operand from A_4 and picking the right operand from A_5, resulting in the program set $\{0 + 1, 0 + y, x + 1, x + y\}$.
- A *union* node is marked by the bold union symbol $\boldsymbol{\cup}$, such as node A_1. It represents the union set of all programs represented by its children.

The key idea of VSA-based synthesis is to construct a VSA for the set of valid programs satisfying all examples. A typical VSA-based synthesizer constructs such a VSA in two steps, by first constructing an individual VSA for each example and then merging these VSAs by intersection.

Single-example VSAs. This paper uses off-the-shelf methods to construct single-example VSAs [Polozov and Gulwani 2015; Wang 2019]. Their result can be viewed as an augmentation of the initial grammar, where each VSA node corresponds to a non-terminal and a specific output value, representing the set of programs that (1) are expanded from this non-terminal, and (2) outputs this value on the example. Fig. 2 shows the single-example VSAs in our sample task. Among them, node A_4 represents the programs that are expanded from non-terminal L and output 0 on example e_a, corresponding to program set $\{0, x\}$. Specially, when there are multiple ways to construct programs



(a) The intersected VSA for examples e_a and e_b. (b) The intersected VSA for all examples.

Fig. 3. The intersected VSAs in our sample task. Here, we use blue to mark the corresponding node pairs (or tuples) in the Cartesian product of single-example VSAs.

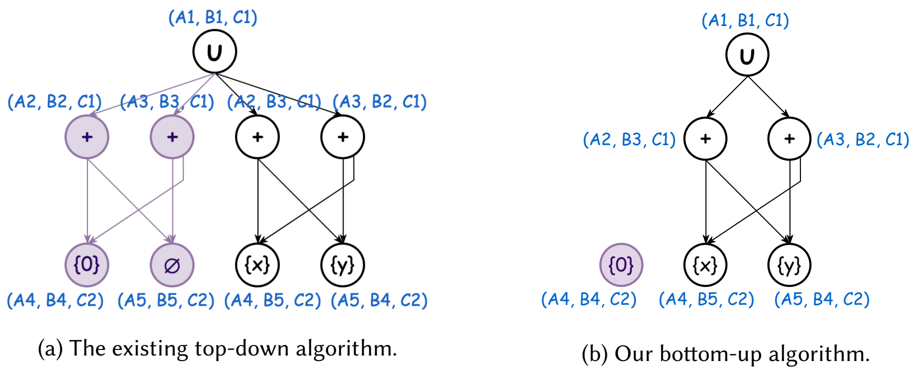
with the same output for a certain non-terminal, such as nodes A_2 and A_3 (two additions with arguments swapped), the VSA will comprise an additional union node, which is node A_1 here, to merge all these possibilities.

Intersecting VSAs. Existing VSA-based synthesizers intersect single-example VSAs by iteration. They first intersect the VSAs for e_a and e_b, resulting in Fig. 3a, and then intersect the result with the VSA for e_c, resulting in Fig. 3b. This final VSA represents two programs, $x + y$ and $y + x$; both of them are valid for our synthesis task.

The existing algorithm for VSA intersection works by traversing the Cartesian product of the input VSAs from the top down [Kini and Gulwani 2015]. For example, to construct the intersected VSA in Fig. 3a, the algorithm first creates the root node with mark (A_1, B_1), the root pair of the input VSAs, which represents the intersection of the program sets of A_1 and B_1. Then, the next step is to decide the node type of (A_1, B_1) and its children. Since both A_1 and B_1 are union nodes, by the equality of $(\bigcup_i S_i) \cap (\bigcup_i T_i) = \bigcup_{i,j}(S_i \cap T_j)$, the intersected program set is equal to the result of first taking the intersection for each children pair and then taking the union. Hence, the top-down algorithm sets the root to a union node and constructs 4 children for it, (A_2, B_2), (A_2, B_3), (A_3, B_2), and (A_3, B_3), each a children pair of A_1 and B_1. After that, the algorithm traverses down to these children and recursively decides their type and children, until reaching the bottom.

In this procedure, the top-down algorithm may construct unnecessary nodes that do not contribute to any program represented by the root. For example, when intersecting the VSAs in Fig. 3a and Fig. 2c, this algorithm will construct nodes (A_2, B_2, C_1) and (A_3, B_3, C_1) as the children of the root, which are not needed, as shown in Fig. 3b. To exclude these nodes, the top-down algorithm post-processes its raw result and takes only the necessary parts.

Efficiency Issue. Despite many known advantages, one major shortcoming of VSA-based synthesis is its inefficiency in processing many examples. This issue originates from the iterative approach for intersecting VSAs. Specifically, after each intersection, the resulting VSA can be much larger than the individual ones because it is constructed over the Cartesian product. For example, the intersected VSA for examples e_a and e_b (Fig. 3a) comprises 9 nodes, larger than the single-example VSAs (Fig. 2a and Fig. 2b), each comprising only 5 nodes. Such an effect can accumulate during the iteration, leading to a giant VSA that exceeds the resource limit.



(a) The existing top-down algorithm.

(b) Our bottom-up algorithm.

Fig. 4. The process of the two intersection algorithms when intersecting the three individual VSAs in our sample task (Fig. 2). Here, we use blue to mark the corresponding node tuples in the Cartesian product, and use violet to mark the unnecessary nodes introduced by each algorithm.

2.2 Multi-Way Intersection

This paper aims to improve the efficiency of VSA intersection. To begin with, we observe that the final intersection is usually small in practice. In our sample task, the final VSA (Fig. 3b) uses only 5 nodes, smaller than the intermediate one (Fig. 3a). This is because a PBE task usually comprises enough IO examples to ensure a small set of valid programs; otherwise, the synthesizer can hardly pick the desired program from a massive collection of valid choices. For example, in our sample task, without the third example e_c, incorrect programs $0 + 1$ and $1 + 0$ will be mistaken as valid, leading to the risk of an incorrect result. With enough examples, the final VSA needs only to represent a small set of programs and will not take up too many nodes.

Motivated by this observation, our idea is to construct the final VSA by intersecting multiple VSAs at once, for example, directly from the three VSAs in Fig. 2 to the final one (Fig. 3b). In this way, we can avoid the intermediate VSAs, creating great potential for optimization.

To realize such potential, the key is to find an efficient algorithm for intersecting multiple VSAs. However, the previous top-down algorithm is far from efficient enough. It often constructs an enormous number of unnecessary nodes when intersecting multiple VSAs, causing a significant waste of computation. Fig. 4a shows its process when intersecting the three single-example VSAs in Fig. 2a-Fig. 2c. After creating the root (A_1, B_1, C_1), the top-down algorithm identifies 4 possible children, each corresponding to a tuple comprising a child of A_1, a child of B_1, and C_1 itself. However, this algorithm cannot tell which children are necessary for the final VSA, so it has to construct all of them and their descendants. As a result, it constructs 4 unnecessary nodes during the intersection. On the one hand, nodes (A_2, B_2, C_1), (A_3, B_3, C_1), and (A_5, B_5, C_2) represent an empty set and thus are trivially useless; on the other hand, although there is a program in node (A_4, B_4, C_2), this program is not used by the root because both parents of this node are empty. These unnecessary nodes make the raw result as large as the intermediate VSA during the iteration (Fig. 3a), completely offsetting the advantage of multi-way intersection.

2.3 Bottom-Up Intersection

To improve the top-down algorithm, we notice that most of its unnecessary nodes are empty nodes that represent no program. On our sample task (Fig. 4a), such empty nodes include (A_2, B_2, C_1), (A_3, B_3, C_1), and (A_5, B_5, C_1), accounting for 75% of the unnecessary nodes. Although these empty nodes are trivially useless, the top-down algorithm cannot be aware of them because of its top-down

Table 1. Nodes in the individual VSAs that are related to the leaf nodes in the intersection (Fig. 4b).

VSA	Related Nodes	Related Parent Nodes	
Fig. 2a	$\{A_4, A_5\}$	$A_2 \to A_4 + A_5$	$A_3 \to A_5 + A_4$
Fig. 2b	$\{B_4, B_5\}$	$B_2 \to B_4 + B_5$	$B_3 \to B_5 + B_4$
Fig. 2c	$\{C_2\}$	$C_1 \to C_2 + C_2$	

nature. Specifically, whether a node is empty is a bottom-up property decided by its descendants; for example, the emptiness of node (A_2, B_2, C_2) is caused by its empty right child (A_5, B_5, C_2). In contrast, the top-down algorithm constructs every node before its descendants. Therefore, it cannot check the emptiness on time and has to construct all visited nodes.

To address this issue, we design a bottom-up algorithm to avoid empty nodes. Fig. 4b shows its process on our sample task. Our algorithm starts with the non-empty leaf nodes. To find them, it searches for the programs that are shared among the leaf nodes of all individual VSAs; here are 3 such programs, 0 shared by (A_4, B_4, C_2), x shared by (A_4, B_5, C_2), and y shared by (A_5, B_4, C_2). For each program, our algorithm constructs a leaf node for the corresponding node tuple and adds the program to the labeled program set. In this way, it constructs 3 leaf nodes for the intersected VSA, all of which are ensured to be non-empty.

Then, we move up to construct higher nodes. Our key idea is to construct new nodes only from existing non-empty nodes, thus ensuring their non-emptiness. Currently, only the three leaf nodes are known non-empty. To construct new nodes from them, our algorithm first collects their related nodes in individual VSAs and all parents of these related nodes, as shown in Tab. 1.

Our algorithm searches to combine these parent nodes into node tuples, resulting in two new non-empty nodes for the intersected VSA:

$$(A_3, B_2, C_1) \to (A_5, B_4, C_2) + (A_4, B_5, C_2) \qquad (A_2, B_3, C_1) \to (A_4, B_5, C_2) + (A_5, B_4, C_2)$$

Note that these parent nodes have two other possible combinations, as listed below. They are ignored because they use an unavailable node (A_5, B_5, C_2) as a child, marked as blue.

$$(A_2, B_2, C_1) \to (A_4, B_4, C_2) + (A_5, B_5, C_2) \qquad (A_3, B_3, C_1) \to (A_5, B_5, C_2) + (A_4, B_4, C_2)$$

In practice, the number of such invalid combinations can be very large. So we develop an efficient search method based on the *trie* data structure [Briandais 1959] to skip these invalid combinations early. We will introduce its details in Sec. 4.2.

After constructing (A_3, B_2, C_1) and (A_2, B_3, C_1), there will be a new chance to construct higher nodes, so our algorithm will repeat the above procedure until no new node is found. As shown in Fig. 4a, our algorithm constructs only 1 unnecessary node on our sample task, much fewer than the top-down algorithm; on the other hand, our result is smaller than the intermediate VSA during the iteration (Fig. 3a), thus realizing the advantage of multi-way intersection.

3 PROBLEM DEFINITION AND ANALYSIS

This section defines the problem of multi-way intersection and analyzes the previous top-down algorithm for VSA intersection on this problem.

3.1 Multi-Way Intersection Problem

Let us start with a formal definition of *version space algebra (VSA)*.

Definition 3.1 (VSA). A VSA is a pair $V = \langle N, r \rangle$, where N is a set of nodes defined in Tab. 2 and $r \in N$ denotes the root. Each node $n \in N$ can be interpreted into a program set, denoted as $\mathbb{P}(n)$, and the program set represented by the VSA, denoted as $\mathbb{P}(V)$, is defined as the set of its root.

Table 2. The definition of VSA nodes. Each node is in one the following three forms.

Form	Syntax	Interpretation to Program Sets		
Union	$\cup(n_1,\dots,n_k)$	$p \in \mathbb{P}(\cup(n_1,\dots,n_k))$	**if**	$\exists i, p \in \mathbb{P}(n_i)$
Join	$f(n_1,\dots,n_k)$	$p \in \mathbb{P}(f(n_1,\dots,n_k))$	**if**	$p = f(p_1,\dots,p_k)$ **and** $\forall i, p_i \in \mathbb{P}(n_i)$
Leaf	$\{p_1,\dots,p_k\}$	$p \in \mathbb{P}(\{p_1,\dots,p_k\})$	**if**	$\exists i, p = p_i$

p_i denotes a program, n_i denotes a VSA node, and f denotes an operator with the arity k.

Table 3. The injection φ for defining the intersection of VSAs. Its definition follows the first matched case from top down, modulo symmetry.

$n \in N_1$	$n' \in N_2$	The definition of $\varphi(n, n')$		
$\cup(n_1,\dots,n_k)$	$\cup\left(n'_1,\dots,n'_{k'}\right)$	$\cup\left(\varphi(n_i, n'_j)\right)$	**for**	$i \in [1,k]$ **and** $j \in [1,k']$
$\cup(n_1,\dots,n_k)$	n'	$\cup(\varphi(n_i, n'))$	**for**	$i \in [1,k]$
$f(n_1,\dots,n_k)$	$f\left(n'_1,\dots,n'_k\right)$	$f(\varphi(n_i, n'_i))$	**for**	$i \in [1,k]$
$f(n_1,\dots,n_k)$	$g\left(n'_1,\dots,n'_{k'}\right)$	$\emptyset$		
$\{p_1,\dots,p_k\}$	n'	$\{p_i\}$	**for**	$i \in [1,k]$ **and** $p_i \in \mathbb{P}(n')$

For simplicity, this paper considers only acyclic VSAs that represent a finite set of programs. In most program synthesis tasks, an infinite program space can be truncated into finite by setting up a size/depth limit, which increases iteratively until a valid program is found.

We define the intersection of VSAs over the Cartesian product of their node sets, in the same way as the previous studies [Polozov and Gulwani 2015]. Each node in the intersection corresponds to a tuple of nodes from the individual VSAs and represents the programs shared by these nodes. Below, we first define the intersection for two VSAs, and then generalize it to more VSAs.

Definition 3.2 (VSA Intersection). Given two VSAs $V_1 = \langle N_1, r_1\rangle$ and $V_2 = \langle N_2, r_2\rangle$, define their intersection $V_1 \cap V_2$ as a VSA $\langle N, r\rangle$, where

- The node set N is defined as the image of an injection φ from the Cartesian product of the individual node sets, i.e., $N_1 \times N_2$. The definition of φ is shown in Tab. 3. It is naturally derived from the invariant that $\mathbb{P}(\varphi(n_1, n_2)) = \mathbb{P}(n_1) \cap \mathbb{P}(n_2)$, i.e., the node corresponding to pair (n_1, n_2) represents the set of programs shared between n_1 and n_2.
- The root r is defined as $\varphi(r_1, r_2)$, the node corresponding to the pair of the individual roots.

Given a set of VSAs $\mathbb{V}$, define their intersection $\bigcap_{V\in\mathbb{V}} V$ as $\left(\bigcap_{V\in\mathbb{V}/\{V'\}} V\right) \cap V'$ for an arbitrary V' in $\mathbb{V}$, i.e., the result of first taking the intersection for the set except one last VSA and then intersecting the result with the last one. Note that the order of intersection is not important here — the resulting VSA is always isomorphic, whatever the choice of V' is.

Theorem 3.3 (Soundness of VSA Intersection). *For any set of VSAs $\mathbb{V}$, their intersection represents the set of programs shared among all individual VSAs, or formally, $\mathbb{P}\left(\bigcap_{V\in\mathbb{V}} V\right) = \bigcap_{V\in\mathbb{V}} \mathbb{P}(V)$.*

Although Def. 3.2 provides a sound construction, it covers the whole Cartesian product and thus is unnecessarily large in most cases. For example, consider the three individual VSAs in our motivating example (Fig. 2). Their Cartesian product comprises $5 \times 5 \times 2 = 50$ node tuples, but only 5 among them contribute to the final intersection (Fig. 3b). To take out the necessary part in the raw intersection, we further introduce the concept of the *core* of a VSA, which includes only those nodes that contribute to representing at least one program of the root.

Algorithm 1: the top-down algorithm.

```
Input: a set of VSAs $\mathbb{V} = \{V_1, \dots, V_k\}$.
Output: the core intersection of $\mathbb{V}$.
1  $r \leftarrow \varphi(r_1, \dots, r_k)$ where $r_i$ is the root of $V_i$;
2  $N \leftarrow \emptyset$;
3  traverse($r$);
4  return core($\langle N, r \rangle$);
5  Function traverse($n$):
6      if $n \notin N$ then
7          $N \leftarrow N$.add($n$);
8          createChildren($n$);
9          traverse($n'$) foreach $n'$ as a child of $n$;
10     end
```

Definition 3.4 (VSA Core). Given a VSA $V = \langle N, r \rangle$, define the predicate $\mathsf{contribute}_V(n)$ for node $n \in N$ as the minimal predicate satisfying the following constraints:

$$\mathsf{contribute}_V(r) \wedge \forall n. \left(\begin{array}{l} \mathsf{contribute}_V(n) \rightarrow \\ \qquad \forall n' \text{ as a child of } n, \left(\mathbb{P}(n') \text{ is not empty} \rightarrow \mathsf{contribute}_V(n')\right) \end{array} \right)$$

Then, define the core of V, denoted as $\mathsf{core}(V)$, as the sub-VSA of V that comprises only nodes satisfying $\mathsf{contributive}_V$ and the edges connecting between them.

In this paper, we use multi-way intersection to handle multiple examples in VSA-based synthesis, where the key problem is to compute the core intersection for a set of given VSAs.

Definition 3.5 (Multi-Way Intersection Problem). Given a set $\mathbb{V}$ of VSAs, compute $\mathsf{core}(\bigcap_{V \in \mathbb{V}} V)$.

3.2 Top-Down Intersection

Polozov and Gulwani [2015] propose a top-down algorithm for intersecting VSAs[2], as shown in Algorithm 1. The algorithm starts with constructing the root node (Line 1, where φ is the injection in Tab. 3) and then builds its descendants from the top down via function traverse (Lines 5-10). Each time when visiting a new node, the algorithm constructs its children (Line 8) and deals with each child recursively (Line 7). At last, a VSA is constructed using all visited nodes, and its core is returned as the result (Line 4).

However, this algorithm is inefficient for multi-way intersection. It constructs all nodes that are reachable from the root, but the number of such nodes faces a combinatorial explosion as the number of VSAs increases. This issue is particularly significant when the synthesis task involves some special operators, such as commutative and associative ones, as shown in the lemma below.

Lemma 3.6 (Case Study). *Consider the following grammar that involves a constant c, an input variable x, a commutative and associative operator ⊕, and a depth limit of h.*

$$S_i \coloneqq S_{i+1} \oplus S_{i+1} \quad \textbf{\textit{for}}\ i \in [1, h) \qquad\qquad S_h \coloneqq c \mid x$$

Let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the formula below provides a lower-bound for the number of nodes constructed by the top-down algorithm when intersecting $\mathbb{V}$, where $\#jnode(V, i)$ denotes the number of join nodes related to S_i.

$$\textstyle\sum_{i=1}^{h} \prod_{V \in \mathbb{V}} \#jnode(V, i)$$

If we further assume the VSA nodes distribute uniformly at each level, this lower-bound will become $(2h)^{1-|\mathbb{V}|} \prod_{V \in \mathbb{V}} |V|$, which is close to the size of the Cartesian product.

[2]This algorithm is originally proposed for two VSAs, but can be naturally generalized to intersecting multiple ones.

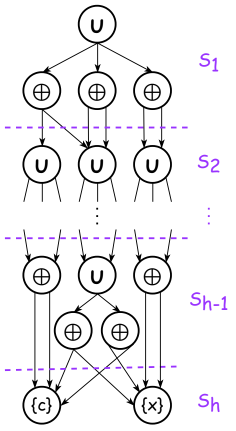


Fig. 5. A sample VSA in Lem. 3.6.

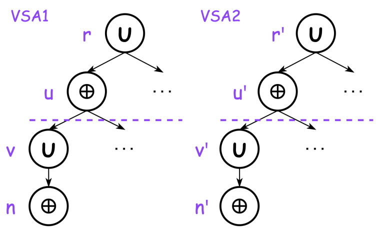


Fig. 6. The position of two join nodes at level 2.

Explanation. Here we provide an informal explanation for this lemma. The formal proofs in this paper can be found in the supplementary material.

As mentioned in Sec. 2.1, the existing approach for constructing single-example VSAs works by attaching a possible output to each non-terminal and adding edges according to the semantics of operators. Hence, every VSA in this lemma must follow a h-level structure, as shown in Fig. 5, where each level corresponds to a non-terminal, each join node combines the nodes at a lower level, and each union node merges those join nodes corresponding to the same output. The root at level 1 corresponds to the expected output in the example, while the leaf nodes at level h denote the constant and the input variable.

Our lower-bound is derived from a key observation:

- *when intersecting VSAs with the above structure, the top-down algorithm will construct a node for every tuple of join nodes at the same level.*

To see this point, let us consider the case with two VSAs, and further consider two arbitrary join nodes (n, n') at level 2, as shown in Fig. 6. These two nodes must belong to the left branch of some join nodes u/u' at level 1 because of the commutativity of $\oplus$. In more detail, n belongs to the VSA only when there is a valid program $p_1 \oplus p_2$ where either p_1 or p_2 belongs to $\mathbb{P}(n)$. For the case of p_1, n will be on the left branch; otherwise, there must be another valid program $p_2 \oplus p_1$ because v is commutative, where v appears at the left. Following the structure in the figure, the top-down algorithm will construct nodes for pairs (r, r'), (u, u'), (v, v'), and finally (n, n') in order. This analysis can be naturally generalized to more VSAs and other depths by induction. □

Although practical grammars are more complex, they use commutative and associative operators everywhere, such as + in integer arithmetic, and/or in the Boolean domain, bvxor/bvmul in bit-vector manipulation, and the index shifting operator in string processing. When intersecting VSAs on these grammars, their sub-structure formed by commutative and associative operators will still explode, as shown in Thm. 3.7. For example, suppose a commutative and associative operator contributes 100 nodes at a fixed level in each VSA. Then, the top-down algorithm will construct at least 100^4 nodes when intersecting 4 such VSAs, causing a significant overhead.

Theorem 3.7. *Let G be any grammar involving commutative and associative operators, and let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the following formula provides a lower-bound for the number of constructed nodes when invoking the top-down*

Algorithm 2: our bottom-up algorithm, where φ is the injection mapping a node tuple to a node in the intersection (Tab. 3).

```
Input: a set of VSAs $\mathbb{V} = \{V_1, \dots, V_k\}$.
Output: the core intersection of $\mathbb{V}$.
1  $N \leftarrow \{\}$;
2  Function construct($n_1, \dots, n_k$):
3      $node \leftarrow \varphi(\overline{n_i})$;
4      if $node \notin N$ then $N \leftarrow N$.add($node$);
5  foreach program $p$ represented by a leaf in $V_1$ do
6      $L_i \leftarrow$ the set of leafs in $V_i$ that represent $p$;
7      construct($\overline{n}$) foreach $\overline{n} \in L_1 \times \cdots \times L_k$;
8  end
9  repeat buildUp($N$) until no new node is added;
10 addEdgeAmong($N$);
11 $r \leftarrow \varphi(\overline{r_i})$ where $r_i$ is the root of $V_i$;
12 return core($\langle N, r \rangle$);
13 Function buildUp($N_{pre}$):
14     foreach operator $f$ do
15         $J_i \leftarrow$ the set of $f$-labeled join nodes in $V_i$ of
             which every child is used by some node
             tuple in $N_{\mathrm{pre}}$;
16         foreach $\overline{n} \in$ combine($f, \overline{J_i}, N_{pre}$) do
17             construct($\overline{n}$);
18         end
19     end
20     foreach $\varphi(n_1, \dots, n_k) \in N_{pre}$ do
21         $U_i \leftarrow$ the set of union nodes in $V_i$ that uses
             $n_i$ as a child;
22         $U_i \leftarrow \{n_i\}$ if $U_i$ is empty;
23         construct($\overline{n'}$) foreach $\overline{n'} \in U_1 \times \cdots \times U_k$;
24     end
```

algorithm on $\mathbb{V}$.

$$\sum_{\oplus \in G} \sum_{i>0} \prod_{V \in \mathbb{V}} \#jnode(V, \oplus, i)$$

where $\oplus$ ranges over the commutative and associative operators in G, and $\#jnode(V, \oplus, i)$ denotes the number of join nodes in V reachable from the root by passing only union nodes and exactly i $\oplus$-labeled join nodes.

Please note that commutativity and associativity are not the only factors making the top-down algorithm inefficient. Many other operators, such as if-then-else, can also cause an exponential increase in the number of constructed nodes. Here we take commutative and associative operators as an example because they are the most common in practice and also the simplest to analyze.

4 APPROACH

This section introduces our bottom-up intersection algorithm (Sec. 4.1 and Sec. 4.2), analyzes its properties (Sec. 4.3), and at last introduces the full synthesis approach (Sec. 4.4).

4.1 Bottom-Up Intersection

The key idea of our bottom-up algorithm is to construct new nodes only from the lower non-empty nodes, thus avoiding the unnecessary nodes that represent no program. Algorithm 2 shows the pseudocode of our algorithm. It first constructs non-empty leaf nodes in the intersection (Lines 5-8), then iteratively builds non-empty join/union nodes via function buildUp (Lines 9 and 13-24), and at last constructs the core intersection from all existing nodes (Lines 10-12).

Leaf nodes. For each program represented by a leaf node in V_1 (Line 5), our algorithm collects all other nodes representing this program (Line 6) and constructs the tuples formed by these nodes (Line 7). Any such tuple must correspond to a non-empty node that at least represents program p.

Join nodes. Given a set N_{pre} of the existing nodes (Line 13) and an operator f with the arity t (Line 14), the first part of buildUp constructs all f-labeled join nodes whose children all belong to N_{pre}

Table 4. The node sets in Example 4.1.

Set	Join Nodes	
J_1	$A_2 : A_4 + A_5$	$A_3 : A_5 + A_4$
J_2	$B_2 : B_4 + B_5$	$B_3 : B_5 + B_4$
J_3	$C_1 : C_2 + C_2$	

Table 5. The node sets in Example 4.2.

Set	Union Node
U_1	$A_1 : \cup(A_2, A_3)$
U_2	$B_1 : \cup(B_2, B_3)$
U_3	$C_1 : C_2 + C_2$

(Lines 15-18). Formally, such nodes correspond to the node tuples in the following form.

$$\left(n'_1, \dots, n'_k\right) \quad \textbf{where} \quad \forall i \in [1,k], n'_i \text{ is a join node } f\left(n'_{i,1}, \dots, n'_{i,t}\right) \in V_i$$
$$\textbf{and} \quad \forall j \in [1,t], \varphi\left(n'_{1,j}, \dots, n'_{k,j}\right) \in N_{\text{pre}}$$

To find such tuples, our algorithm collects all possible n'_i into set J_i (Line 15) and uses function combine to combine these sets into node tuples that satisfy the above condition (Line 16). We will introduce the implementation of combine later in Sec. 4.2.

Example 4.1 (Join). Consider the first invocation of buildUp in our motivating example (Fig. 4b), where the input set N_{pre} comprises the three non-empty leaf nodes, as shown below:

$$\{\varphi(A_4, B_4, C_2), \quad \varphi(A_4, B_5, C_2), \quad \varphi(A_5, B_4, C_2)\}$$

When operator + is picked at Line 14 to construct new join nodes, node sets J_i will include the nodes listed in Tab. 4, and the corresponding result of combine will be $\{(A_2, B_3, C_1), (A_3, B_2, C_1)\}$.

Union nodes. Then, the second part of buildUp constructs the union nodes that have a child in N_{pre} (Lines 20-24). Such nodes correspond to the node tuples in the following form.

$$\left(n'_1, \dots, n'_k\right) \quad \textbf{where} \quad \forall i \in [1,k], n'_i \text{ is a union node in } V_i$$
$$\textbf{and} \quad \exists \varphi(n_1, \dots, n_k) \in N_{pre}, \forall i \in [1,k], n_i \text{ is a child of } n'_i$$

To find such tuples, our algorithm first picks a node tuple $(n_1, \dots, n_k)$ from N_{pre} (Line 20), then collects all possible n'_i into set U_i (Line 21), and constructs the entire Cartesian product of these sets (Line 23). Specially, node n_i will be added to U_i when it has no union parent (Line 22), as each node can be regarded as a union node comprising itself.

Example 4.2 (Union). In our motivating example (Fig. 4b), when node $\varphi(A_2, B_3, C_1)$ is picked at Line 20 to construct new union nodes, node sets U_i will include the nodes listed in Tab. 5, where C_1 appears in U_3 because it does not have any union parent. From these sets, our algorithm will find node $\varphi(A_1, B_1, C_1)$, the root of the intersected VSA[3].

Data structures. Note that Algorithm 2 shows only the key idea of our bottom-up algorithm. Its efficiency also relies on several data structures not yet mentioned, for example, to maintain the node set N (Lines 4) and to construct the sets J_i and U_i (Lines 15 and 21). These details are omitted here for simplicity, and can be found in Appendix B.

4.2 Trie-Based Combination

Now we go into the details of the function combine (Line 16) for constructing the join nodes.

- The input of combine includes an operator f, several node sets $J_1, \dots, J_k$, each comprising some f-labeled join nodes in an input VSA, and a set N of nodes in the intersection.

[3]The reader may notice that our algorithm does not construct all non-empty union nodes. For example, node $\varphi(A_2, B_1, C_1)$ is a parent of $\varphi(A_2, B_2, C_1)$ by Def. 3.2 and represents programs $x + y$ and $y + x$, but it is skipped. The details and the soundness of this optimization can be found in Appendix B.

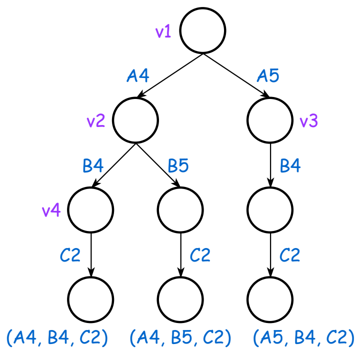


Fig. 7. A trie built for keys (A_4, B_4, C_2), (A_4, B_5, C_2), and (A_4, B_5, C_2), where we use blue to mark tokens and keys, and use violet to assign indices for some nodes.

Algorithm 3: function combine.

```
Input: operator $f$, node sets $J_1, \dots, J_k, N$.
Output: the set of valid node tuples.
1  $root \leftarrow$ buildTrie($N$); $res \leftarrow \emptyset$;
2  Function search($m, \overline{n}, u_1, \dots, u_t$):
3      if $m = k$ then $res \leftarrow res$.add($\overline{n}$);
4      else foreach $n_{m+1} \in J_{m+1}$ do
5          $n_{m+1,1}, \dots, n_{m+1,t} \leftarrow$ children of $n_{m+1}$;
6          $u'_j \leftarrow$ moveDown($u_j, n_{m+1,j}$) for $j \in [1, t]$;
7          if no $u'_j$ is null then
8              search($m + 1, \overline{n}$.add($n_{m+1}$), $u'_1, \dots, u'_t$);
9          end
10     end
11 search(0, (), $root, \dots, root$);
12 return $res$;
```

- Its goal is to search among the Cartesian product of J_i and return those node tuples whose children all belong to N. Formally, it needs to return all tuples $(n_1, \dots, n_k)$ in $J_1 \times \cdots \times J_k$ satisfying the following condition, where t denotes the arity of f.

$$\forall j \in [1, t], \varphi\big(n_{1,j}, \dots, n_{k,j}\big) \in N \quad \textbf{where } \forall i \in [1, k], f\big(n_{i,1}, \dots, n_{i,t}\big) = n_i \tag{1}$$

A direct implementation of this function is to build the entire Cartesian product and then filter out the valid tuples using the condition. This implementation, however, is inefficient because the size of the Cartesian product is typically much larger than the number of valid tuples, resulting in a significant waste on processing invalid tuples. To address this issue, our implementation employs the *trie* data structure [Briandais 1959] to skip the invalid tuples early.

A trie is a search tree built for a set of keys, each is a sequence of tokens. As illustrated in Fig. 7, each trie edge is labeled with a token, and each trie node represents the sequence of tokens on the path from the root to it. Given a set of keys, the corresponding trie comprises only the nodes that represent the keys and their ancestors. Hence, there is a one-to-one correspondence between the trie nodes and all prefixes of the keys.

Switching back to the function combine, suppose the first m nodes $n_1, \dots, n_m$ have been picked from the first m sets, while the last $k - m$ nodes are still undecided. At this time, the assignments to $n_{i,j}$ in Eq. 1 are fixed for $i \in [1, m]$ and $j \in [1, t]$, so the condition cannot be true if there is an index j such that $(n_{1,j}, \dots, n_{m,j})$ is not a prefix of any tuple in N. In other words, the following formula is a sufficient condition for ¬Eq. 1.

$$\exists j \in [1, t], \forall \varphi(\overline{n'}) \in N, \quad \big(n_{1,j}, \dots, n_{m,j}\big) \text{ is not a prefix of } \overline{n'} \tag{2}$$

Our implementation of combine uses this formula as a pruning condition and builds a trie to perform the prefix check, as shown in Algorithm 3. It builds a trie for node tuples in N (Line 1) and searches among the Cartesian product via function search (Lines 2-11). In the input of search, m denotes the number of nodes that have been decided, $\overline{n}$ stores the assignments to the first m nodes, and u_j for $j \in [1, t]$ denotes the trie node corresponding to sequence $(n_{1,j}, \dots, n_{m,j})$.

Initially, all nodes are unknown; so m is 0, $\overline{n}$ is an empty tuple, and u_i are all the root node of the trie (Line 11). Then, in each step, search picks a node from the next node set J_{m+1} (Line 4) and updates the trie nodes via function moveDown (Lines 6-7). This function returns the child of u_j that is linked with label $n_{m+1,j}$, and will return an error token *null* if such a child does not exist.

After the update, function search will skip the current search branch if any of the new trie nodes is *null* (Line 7) since the pruning condition Eq. 2 is satisfied. Otherwise, it will move to enumerate the next node recursively (Line 8).

Example 4.3. In Example 4.1, function combine is invoked with f as +, set J_i as listed in Tab. 4, and set N comprising $\varphi(A_4, B_4, C_2)$, $\varphi(A_4, B_5, C_2)$, and $\varphi(A_5, B_4, C_2)$. At this time, the trie is shown as Fig. 7, and one search trace of search is as follows.

(1) The initial invocation at Line 11 is search$(0, (), v_1, v_1)$.
(2) Node $A_2 : A_4 + A_5$ is picked from J_1. Then, the first trie node is moved from v_1 to v_2 through label A_4, and the second is moved from v_1 to v_3 through label A_5; so the next invocation is search$(1, (A_2), v_2, v_3)$.
(3) Node $B_2 : B_4 + B_5$ is picked from J_2. Then, the first trie node is moved from v_2 to v_4 through label B_4, and the second node v_3 becomes *null* because v_3 has no outgoing edge labeled with B_5. Hence, search will ignore this branch and thus skip the invalid tuple (A_2, B_2, C_2).

4.3 Properties

4.3.1 Soundness. Our algorithm can correctly construct the core intersection of VSAs (Thm. 4.4).

THEOREM 4.4 (SOUNDNESS). *Given a set $\mathbb{V}$ of VSAs, the output of Algorithm 2 must be equal to the core intersection core$(\bigcap_{V \in \mathbb{V}} V)$.*

4.3.2 Efficiency. Before an empirical evaluation, we first study the efficiency of our algorithm by theoretically comparing it with existing methods.

Comparing with the top-down algorithm. Continuing with the case study in Lem. 3.6, we prove that our algorithm can efficiently handle commutative and associative operators (Lem. 4.5). Under the same setting as Lem. 3.6, the number of nodes constructed by our algorithm is always polynomial in the depth h and the size of programs; this number is significantly smaller than the exponential counterpart of the top-down algorithm, especially when given many VSAs.

LEMMA 4.5 (CASE STUDY). *Consider the grammar in Lem. 3.6, repeated as follows, where c is a constant, x is the input variable, $\oplus$ is a commutative and associative operator, and h is a depth limit.*

$$S_i \coloneqq S_{i+1} \oplus S_{i+1} \quad \textbf{\textit{for}}\ i \in [1, h) \qquad\qquad S_h \coloneqq c \mid x$$

Let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed from a given IO example, and let p^* be the largest program represented by any VSA in $\mathbb{V}$. Then, Algorithm 2 will construct at most $h \cdot \mathit{size}^2(p^*)$ nodes when intersecting the VSAs in $\mathbb{V}$.

Comparing with observational equivalence. Another method worth comparing is *observational equivalence (OE)* [Udupa et al. 2013], a pruning strategy for avoiding duplicated programs in bottom-up enumeration. It will evaluate each visited program on the example set, record the outputs, and skip all programs whose outputs are duplicated. This method is popular because of its *generality* — it relies only on operator-level semantics, hence can be applied to most synthesis problems.

In this sense, whether there is a general synthesizer that is more efficient than OE becomes an interesting question and has not been answered before; for example, although VSA-based synthesis has achieved the same level of generality and better practical performance in several domains [Wang et al. 2017], this advantage is not guaranteed because of the inefficient intersection (i.e., Lem. 3.6).

We prove that our intersection algorithm has bridged this last gap. As shown in Thm. 4.6, when exploring the same program space, the number of VSA nodes constructed by our algorithm is always no larger than the number of programs constructed by OE, within a constant factor.

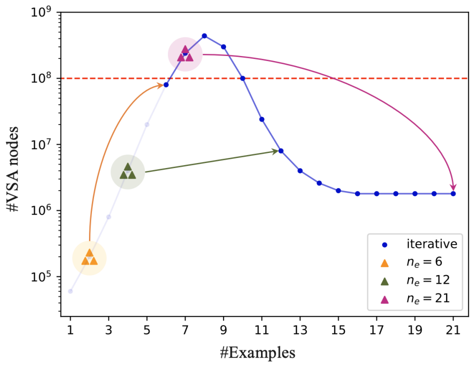


Fig. 8. The comparison of using different numbers of examples in multi-way intersection, where 3 individual VSAs are used. In this figure, n_e denotes the number of examples in multi-way intersection, the dotted line denotes the resource limit, triangles denote the size of individual VSAs, and arrows denote the process of intersecting VSAs.

Algorithm 4: Our PBE solver.

```
Input: A PBE task $T$ with examples $\{e_1, \dots, e_m\}$,
       a VSA builder $\mathcal{B}$ for single-example
       VSAs, and a parameter $n_v$.
Output: A program satisfying all examples.
$V_i \leftarrow \mathcal{B}(T, e_i)$ for $i \in [1, n_v]$;
$n_e \leftarrow n_v$;
while $\neg$isEnough$(T, V_1, \dots, V_{n_v})$ do
    foreach $i \in [1, n_v]$ do
        $V_i \leftarrow$ intersect$(V_i, \mathcal{B}(T, e_{n_e+i}))$;
    end
    $n_e \leftarrow n_e + n_v$;
end
$V \leftarrow$ multiIntersect$(V_1, \dots, V_{n_v})$;
foreach $i \in [n_e + 1, m]$ do
    $V \leftarrow$ intersect$(V,$ buildVSA$(T, e_i))$;
end
return any program in $V$;
```

Theorem 4.6 (Worst-Case Efficiency). *The following inequality always hold for any finite program space P and any example set E.*

$$cost_{Ours}(\mathbb{V}) \leq 2cost_{OE}(P, E)$$

where

- *$\mathbb{V}$ is the set of single-example VSAs constructed over program space P and example set E.*
- *$cost_{Ours}(\mathbb{V})$ denotes the number of VSA nodes constructed by our bottom-up algorithm when intersecting the VSAs in $\mathbb{V}$.*
- *$cost_{OE}(P, E)$ denotes the number of programs constructed by OE after traversing the whole program space P with the example set E.*

Explanation. Recall that our bottom-up algorithm constructs only non-empty VSA nodes. Let v be an node it constructs, and let p be a program represented by v. Since p is in the program space P, OE must have recorded the outputs of p after traversing the whole space; hence v can be connected to a program p' constructed by OE whose outputs are the same as p. By properly deciding the choice of p', we prove that this connection can form an at most 2-to-1 mapping from the VSA nodes to the constructed programs, thus obtaining the target theorem. □

Note that the above theorem illustrates only the worst case. In practice, our algorithm typically constructs fewer nodes, thus letting VSA-based synthesis outperform OE significantly. Details on this comparison can be found in Sec. 5.3.

4.4 Multi-Way Interaction for Programming by Example

At the end of this section, we discuss how to integrate the multi-way intersection efficiently into a PBE solver. Our straightforward approach, as in Thm. 4.6, is to construct an individual VSA for each IO example and intersect these VSAs at once.

Grouping strategy. However, this approach can be unnecessarily inefficient when the number of examples is large, because it exposes too many VSAs to the intersection algorithm. Specifically, although our algorithm is optimized for multi-way intersection, its overhead will still increase when its input comprises more VSAs; for example, the combine function (Algorithm 3) searches among the Cartesian product of several node sets, each from a VSA, so this search space can still be large when intersecting too many VSAs.

To address this issue, we propose a *grouping strategy* to limit the number of VSAs. Given a pre-defined parameter n_v and a set of IO examples, this strategy first divides the examples into several groups, constructs an individual VSA for each group, and then intersects all these VSAs simultaneously; in this way, no matter how many examples there are, the intersection algorithm needs only to process n_v VSAs, with limited overhead.

Adaptive grouping. But on the other hand, the grouping strategy introduces a new issue: the individual VSA of a group can grow too large when there are many examples but n_v is too small. This is illustrated by the purple line in Fig. 8, where the individual VSAs appear at the peak; their scale exceeds the resource limit, and their construction becomes a new bottleneck.

To avoid such an extreme case, we propose to *adaptively* decide the scope of multi-way intersection, as shown in Algorithm 4. Our solver starts by initializing each individual VSA with only one example (Line 1) and then enlarges these VSAs iteratively, each time adding one example (Lines 4-6), until a termination condition is satisfied (Line 3). After intersecting these VSAs via our bottom-up algorithm (Line 9), our solver incorporates the remaining examples (Line 10-12) and returns a program in the final VSA as the result (Line 11).

Termination condition. In this algorithm, the termination condition isEnough (Line 3) is crucial to the overall performance. As shown by the yellow line in Fig. 8, if this condition adds too few examples to multi-way intersection, the intersected VSA will lie before the peak, then the largest VSA will still be constructed when processing the remaining examples; on the other hand, if it adds too many examples, the individual VSAs will be large, leading to the same problem as the straightforward grouping strategy. Ideally, the number of examples should be the smallest value to let the intersected VSA appear at the right foot of the hill.

To achieve this ideal case, our isEnough estimates the number of programs in the intersection and returns true once this number falls below a pre-defined threshold, since a VSA comprising few programs must be small. The estimation is defined as follows, where P denotes the program space.

$$|P| \cdot \prod_{i=1}^{n_v} r_i \qquad \textbf{where } r_i \coloneqq \frac{\mathbb{P}(V_i)}{|P|}$$

In this estimation, we first compute the ratio r_i of programs in P that are included in the ith VSA V_i — it denotes the probability for a random program to be in V_i. Then, we naively assume that these probabilities are independent for different VSAs, under which the expected number of programs remaining in the intersection will be the product of all these probabilities and the size of P, as shown in the left formula above.

Note that this estimation is not for *precisely* counting the programs in the intersection; instead, it is designed to efficiently compute a *rough* number, so that our solver can, with a low overhead, find a good point for performing multi-way intersection. The bias in the estimation mainly comes from the independence assumption. In practice, IO examples are mostly dependent, leading to complicated dependencies among VSAs; for example, if an IO example is generated specifically to exclude incorrect programs that satisfy all previous examples, its VSA will be more effective at excluding programs in previous VSAs than a random program in the whole program space. It is possible to design a better estimation to account for such dependencies in the future; however, such an estimation is not the main focus of this paper, and could be overly complicated for our synthesizer, e.g., too complex to be computed efficiently.

5 EVALUATION

We implemented our approach (Algorithm 4) into a tool named MOLE, which can be integrated into any existing VSA-based synthesizers [Polozov and Gulwani 2015; Wang et al. 2017, 2018]. To evaluate its effectiveness, we design experiments to answer the following research questions.

- **RQ1**: How does MOLE perform in a general VSA-based synthesizer?
- **RQ2**: How does MOLE perform in a specialized VSA-based synthesizer?
- **RQ3**: How does each component of MOLE contribute to its performance?

5.1 Implementation

Our implementations are all in C++ and can be found in the supplementary materials.

Synthesizers. We also implement two VSA-based synthesizers by integrating MOLE into existing synthesizers, following Algorithm 4.

The first synthesizer is based on *finite-tree automata* (FTA) [Wang et al. 2017], named MOLE_{FTA}. Given an input-output example, MOLE_{FTA} first grows an FTA as the existing synthesizers [Wang et al. 2017]; then, since this paper considers only finite program spaces, the resulting FTA must be acyclic, so it can be converted into a VSA via the mapping constructed by Koppel [2021]. This synthesizer is general because it requires only the forward interpretation of the operators, hence can be applied to any SyGuS problem [Alur et al. 2013] without further customization.

The second synthesizer is based on BLAZE [Wang et al. 2018], named $\text{MOLE}_{\text{BLAZE}}$, which builds VSAs with abstract values rather than concrete ones to reduce the cost. Compared with the general synthesizer, BLAZE requires an additional universe of predicates to define the abstract space, hence is specialized; on the other hand, it can be much more efficient if a proper abstract space exists, for example, BLAZE is still a SOTA synthesizer over a special program space in the string domain. We implement $\text{MOLE}_{\text{BLAZE}}$ for the string domain under the same setting as BLAZE, i.e., using the same program space and the same universe of predicates.

Parameters. Our approach (Algorithm 4) relies on two parameters, n_v denotes the number of input VSAs for multi-way intersection and n_{lim} in function isEnough denotes the threshold on the number of remaining programs. By default, we set n_v to 3 according to the result of a small-scale experiment, and set n_{lim} to 10^5 because a VSA with no more than 10^5 programs is usually small enough for a quick process.

5.2 Experimental Setup

Baseline solvers. We compare MOLE_{FTA} and $\text{MOLE}_{\text{BLAZE}}$ with their SOTA counterparts.

- For the general synthesizer MOLE_{FTA}, we consider the standard VSA-based synthesizer (RAWVSA) and observational equivalence (OE). RAWVSA constructs single-example VSAs via FTA [Wang et al. 2017] and uses the previous top-down algorithm for intersection [Polozov and Gulwani 2015]; OE enumerates programs from small to large and skips all programs that have duplicated outputs on the example set [Udupa et al. 2013].
- For the specialized synthesizer $\text{MOLE}_{\text{BLAZE}}$, we consider its original synthesizer BLAZE, the SOTA on the program space supported by $\text{MOLE}_{\text{BLAZE}}$. BLAZE uses its VSA builder to build a VSA for all examples at once.

Dataset. We consider three synthesis domains in SyGuS-Comp [Alur et al. 2019], *integer arithmetic*, *cryptographic circuits*, and *string manipulation*. Tab. 6 lists the profile of the datasets we used in our evaluation — they are collected in the following way.

Table 6. The profile of the datasets in our evaluation.

Name	#Tasks	Domain	#Operators
$\mathcal{D}_I$	100	Integer Arithmetic	10
$\mathcal{D}_C$	581	Cryptographic Circuits	4
$\mathcal{D}_S$	205	String Manipulation	16
$\mathcal{D}_{\text{BLAZE}}$	108		3

Table 7. The results of comparing MOLE$_{\text{FTA}}$ with baselines.

Dataset	Solver	#Solved	Time Cost (sec)				#VSA Nodes		
			Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{A}}$	Ratio$_{\text{H}}$	Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{A}}$
$\mathcal{D}_I$	MOLE$_{\text{FTA}}$	41	0.77				3952		
	RAWVSA	24	0.20	0.10	$\times$ 2.09	$\times$ 5.51	8393	1028	$\times$ 8.17
	OE	21	0.98	0.10	$\times$ 9.60	$\times$ 72.07	NA		
$\mathcal{D}_C$	MOLE$_{\text{FTA}}$	579	0.09				4074		
	RAWVSA	505	0.39	0.05	$\times$ 7.36	$\times$ 34.00	39749	2784	$\times$ 14.28
	OE	482	0.61	0.11	$\times$ 5.72	$\times$ 34.27	NA		
$\mathcal{D}_S$	MOLE$_{\text{FTA}}$	146	0.34				178		
	RAWVSA	140	0.30	0.30	$\times$ 1.00	$\times$ 1.01	566	150	$\times$ 3.78
	OE	152	0.32	0.34	$\times$ 0.93	$\times$ 1.09	NA		

- For integer arithmetic, $\mathcal{D}_I$ is the dataset constructed by Ji et al. [2021]. It augments the dataset in SyGuS-Comp with 18 tasks extracting from the synthesis tasks of divide-and-conquer algorithms [Farzan and Nicolet 2017].
- For cryptographic circuits, $\mathcal{D}_C$ is exactly the same as the dataset in SyGuS-Comp.
- For string manipulation, we use two datasets: $\mathcal{D}_S$ is the dataset in SyGuS-Comp, and $\mathcal{D}_{\text{Blaze}}$ is created by Wang et al. [2018] for evaluating BLAZE, which comprises only operators and problems supported by the predicate space of BLAZE.

Since this paper focuses only on finite program spaces and finite example sets, we simplify some tasks in these datasets as follows.

- For a task with an infinite program space, we will truncate the space according to the size of the expected program. In practice, this truncation can be performed adaptively by iteratively enlarging a size limit, until a valid program is found.
- For a task with an infinite example set (e.g., when the specification is given as an SMT formula), we will randomly generate 100 IO examples and replace the original set with the finite list of these random examples.

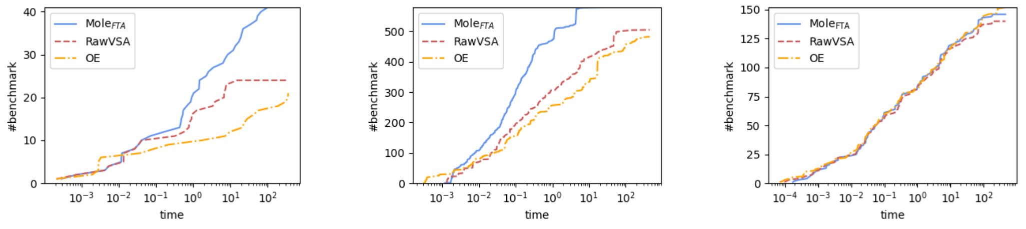
Others. Our experiments are conducted on Intel(R) Xeon(R) Platinum 8369HC 8-Core CPU, with a timeout of 10 minutes and a memory limit of 4GB per task.

5.3 RQ1: Effect in General VSA-Based Synthesis

Setup. We compare MOLE$_{\text{FTA}}$ with RAWVSA and OE on three datasets $\mathcal{D}_I$, $\mathcal{D}_C$ and $\mathcal{D}_S$.

Results. The results are summarized in Tab. 7, organized as follows.

- For each dataset and each solver, column "#Solved" reports the number of solved tasks.



(a) $\mathcal{D}_I$ (b) $\mathcal{D}_C$ (c) $\mathcal{D}_S$

Fig. 9. Number of tasks solved by Mole$_{\text{FTA}}$ and each baseline solver over time on each dataset.

- For each dataset, column "Time Cost" reports the average[4] time cost (in seconds) of each approach and the average acceleration ratio achieved by Mole$_{\text{FTA}}$. We consider only tasks solved by both solvers when computing the averages and ratios. Column "Mean$_{\text{B}}$" reports the average time cost of the baseline approach, while column "Mean$_{\text{O}}$" reports the average time cost of our approach. Column "Mean$_{\text{A}}$" reports the average acceleration ratio achieved by Mole$_{\text{FTA}}$ in comparison with the baseline approach on tasks solved by both approaches, while column "Mean$_{\text{H}}$" further considers only tasks that take at least one solver more than 1 second.
- For each dataset, column "#VSA Nodes" reports the average number of VSA nodes constructed by each approach, and the average saving on this number achieved by Mole$_{\text{FTA}}$. We consider only tasks solved by both approaches in this comparison.

Fig. 9 further shows the number of tasks solved by Mole$_{\text{FTA}}$, RawVSA, and OE over time.

These results demonstrate the effectiveness of Mole$_{\text{FTA}}$. It significantly outperforms RawVSA and OE on $\mathcal{D}_I$ and $\mathcal{D}_C$, by solving more tasks, using less time, and constructing fewer VSA nodes than RawVSA. Such an advantage comes from multiple aspects:

- Compared with RawVSA, Mole$_{\text{FTA}}$ avoids huge intermediate VSAs via multi-way intersection, and avoids wasting time on empty nodes via the bottom-up algorithms.
- Compared with OE, Mole$_{\text{FTA}}$ ensures its VSA nodes are no more than the number of programs enumerated by OE (Thm. 4.6); moreover, this number is typically much smaller because Mole$_{\text{FTA}}$ will skip those programs that are not included in any VSAs.

In contrast to these two datasets, the advantage of Mole is much less significant on $\mathcal{D}_S$. This is because our approach is designed only for the intersection of multiple VSAs, so its advantage can emerge only for tasks requiring multiple examples. In contrast, the IO examples in the string domain are too informative, so that for most tasks in $\mathcal{D}_S$, the program space will be small enough even with only a single example; Mole cannot provide any improvement for RawVSA on such tasks because VSA intersection is no longer the bottleneck.

The same reason also explains why the advantage of Mole tends to be more significant on more challenging tasks (Fig. 9). Such tasks usually require more examples, providing greater room for our method to improve.

5.4 RQ2: Effect in Specialized VSA-Based Synthesis

Setup. We compare Mole$_{\text{Blaze}}$ with Blaze on its dataset $\mathcal{D}_{Blaze}$.

Result. Tab. 8 summarizes the results of this experiment, organized in the same way as Tab. 7. These results show the success of Mole in improving Blaze — Mole further increases the efficiency

[4]We always take the geometric mean when computing the average of multiple results.

Table 8. The results of comparing Mole$_{\text{Blaze}}$ with Blaze.

Dataset	Solver	#Solved	Time Cost (sec)				#VSA Nodes		
			Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{A}}$	Ratio$_{\text{H}}$	Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{B}}$
$\mathcal{D}_{\text{Blaze}}$	Mole$_{\text{Blaze}}$	88	0.02				1466		
	Blaze	86	0.06	0.02	× 2.48	× 1.80	3068	1207	× 2.54

Table 9. The results of comparing Mole$_{\text{FTA}}$ with different variants.

Dataset	Experiment	Solver	#Solved	Time Cost (sec)				#VSA Nodes		
				Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{A}}$	Ratio$_{\text{H}}$	Mean$_{\text{B}}$	Mean$_{\text{O}}$	Ratio$_{\text{B}}$
$\mathcal{D}_C$		Mole$_{\text{FTA}}$	579	0.09				4074		
	Sec. 5.5.1	Mole$_{\text{Top-Down}}$	241	0.03	0.01	× 2.92	× 63.24	3120	624	× 5.00
	Sec. 5.5.2	Mole$_{\text{Iter}}$	579	0.19	0.09	× 2.13	× 3.28	6991	4074	× 1.72
	Sec. 5.5.3	Mole$_{\text{Cart}}$	228	0.16	0.03	× 4.79	× 40.07	9971	1642	× 6.07
	Sec. 5.5.4	Mole$_{\text{All-Once}}$	574	0.38	0.09	× 4.39	× 12.29	11039	3942	× 2.80
		Mole$_{\text{All-Group}}$	579	0.47	0.09	× 5.17	× 8.60	13227	4074	× 3.25
	Sec. 5.5.5	Mole$_{2}$	579	0.09	0.09	× 1.02	× 1.20	4074	4074	× 1.00
		Mole$_{4}$	579	0.09	0.09	× 1.02	× 0.98	4624	4074	× 1.13

and significantly reduces the number of constructed VSA nodes. More generally, such results demonstrate the potential for Mole to improve existing synthesizers. Since existing studies on VSA-based synthesis have primarily focused on the construction of single-example VSAs, Mole offers an orthogonal approach that could provide even more significant speedups when combined with the previous techniques.

5.5 RQ3: Ablation study of each component of Mole

Setup. In this experiment, we conduct ablation studies on different components of Mole. Specifically, we study the effectiveness of our bottom-up intersection algorithm (Algorithm 2), multi-way intersection, trie-based combination (Algorithm 3), and the grouping strategy (Algorithm 4). Besides, we also study the effects of the parameter n_v, the number of VSAs in multi-way intersection.

For simplicity, we consider only one dataset $\mathcal{D}_C$ in the experiment. We chose this dataset because it comprises the most tasks and the advantage of Mole$_{\text{FTA}}$ is most clear; so we expect it to provide the most significant results for the ablation study.

5.5.1 Bottom-up intersection. We consider a variant of Mole$_{\text{FTA}}$, denoted as Mole$_{\text{Top-Down}}$, where the existing top-down algorithm (Algorithm 1) is used to perform all VSA intersection, in place of our bottom-up algorithm.

Result. Tab. 9 summarizes the results of this experiment, organized in the same way as the previous experiments. These results show the effectiveness of our bottom-up algorithm, as the default Mole$_{\text{FTA}}$ significantly outperforms the variant using the top-down algorithm.

Note that Mole$_{\text{Top-Down}}$ performs even worse than the baseline solver RawVSA (Tab. 7), which iteratively intersects VSAs via the top-down algorithm. This result matches our theoretical analysis in Sec. 3.2 — the top-down algorithm is inefficient when intersecting multiple VSAs, making the multi-way intersection even worse than the direct iterative approach.

5.5.2 Multi-way intersection. We consider another variant, denoted as Mole$_{\text{Iter}}$, which uses our bottom-up intersection algorithm to perform iterative intersection.

Result. The results are listed in Tab. 9. They show the effectiveness of our multi-way intersection algorithm, as the default Mole$_{\text{FTA}}$ outperforms the variant using iterative intersection.

5.5.3 *Trie-based combination.* We consider a variant, denoted as $\text{MOLE}_{\text{CART}}$, which builds the entire Cartesian product instead of our trie-based combination (Algorithm 3).

Result. The results are summarized in Tab. 9. They demonstrate the effectiveness of our trie-based combination, which reduces the cost in processing invalid tuples when constructing the join nodes.

5.5.4 *Grouping strategy.* We consider two variants of MOLE_{FTA}.

- $\text{MOLE}_{\text{ALL-ONCE}}$ sets the grouping parameter n_v to the number of provided examples, instead of 3. That is, each group contains only one example.
- $\text{MOLE}_{\text{ALL-GROUP}}$ disables the grouping termination condition function isEnough (Line 3 of Algorithm 4), and uses all provided examples for the grouping intersection process (Line 3-8 of Algorithm 4).

Result. The results are listed in Tab. 9. They show the effectiveness of our grouping strategy, as the default MOLE_{FTA} outperforms each variant. This matches our analysis in Sec. 4.4 that a suitable grouping strategy is required for the best performance.

5.5.5 *Parameter selection.* We consider two variants of MOLE_{FTA}, denoted as MOLE_2 and MOLE_4, with the parameter n_v in multi-way intersection set to 2 and 4 respectively instead of the default 3. .

Result. The results are listed in Tab. 9. They demonstrate that MOLE is robust regarding the choice of n_v, since all three choices of n_v yielded similar performance.

6 RELATED WORK

VSA-based synthesis. VSA has achieved great success in program synthesis [Dong et al. 2022; Gulwani 2011; Li et al. 2024; Padhi et al. 2018; Polozov and Gulwani 2015; Wang 2019; Wang et al. 2017]. A VSA-based synthesizer typically comprises two components, one for constructing single-example VSAs and the other for intersecting these VSAs.

There are two major methods for constructing single-example VSAs, one works from the top down via witness functions [Polozov and Gulwani 2015], and the other works from the bottom up via FTA [Wang 2019]. On top of the FTA-based method, BLAZE [Wang et al. 2018] applies abstraction refinements to further reduce the scale of the resulting VSA. The contribution of this paper is orthogonal to these methods, and our approach MOLE can be combined with any of them.

As for VSA intersection, the previous method uses a top-down algorithm for two VSAs and handles more by iteratively intersecting them one by one [Polozov and Gulwani 2016]. This method is inefficient when processing many VSAs, and this paper addresses this issue by proposing the scheme of multi-way intersection and the bottom-up algorithm for intersecting VSAs.

There are also theoretical results on the hardness of VSA intersection. Kozen [1977] shows this problem is PSPACE-complete, and Karakostas et al. [2003] further prove that there is no algorithm better than building the entire Cartesian product, unless two complexity classes NL and P are different[5]. However, these results do not conflict with our progress in designing an empirically efficient algorithm because they rely on extreme cases that can hardly appear in practice.

Besides, this paper considers only context-free programs spaces, following the SyGuS framework [Alur et al. 2013], while there has been recent work applying VSA-based synthesis to context-sensitive domains, such as recursions [Miltner et al. 2022] and programs with local variables [Li et al. 2024]. It is future work to extend our approach to these domains.

Programming by example. Besides VSA-based approaches, there have been numerous other synthesizers proposed for various scenarios [Alur et al. 2015; Ding and Qiu 2024; Jha et al. 2010; Ji

[5]Whether NL is equal to P is still an open problem after decades of research.

et al. 2021; Lee 2021; Lee et al. 2018; Reynolds et al. 2019; Udupa et al. 2013]. They make trade-offs between keeping general and improving efficient via special domain properties.

Among these synthesizers, *observational equivalence (OE)* [Udupa et al. 2013] achieves the highest level of generality. It relies only on the interpreter of operators, hence can be applied to most synthesis domains. In this paper, our approach MOLE is designed keep the same level of generality. We prove that MOLE performs no worse than OE theoretically, and our evaluation results show that MOLE is significantly more efficient than OE.

The other synthesizers either require more inputs or utilize domain properties to improve the efficiency, such as using efficient witness functions [Cambronero et al. 2023; Lee 2021], relying on constraint solvers [Jha et al. 2010; Ji et al. 2021; Reynolds et al. 2019], applying abstractions [Yoon et al. 2023], incorporating specialized optimizations for certain operators [Alur et al. 2015, 2017; Ding and Qiu 2024], and combining with probabilistic models [Balog et al. 2017; Ji et al. 2020; Lee et al. 2018]. Our general synthesizer MOLE$_{\text{FTA}}$ can hardly outperform these synthesizers. This is not surprising because, in theory, general optimizations should not be more effective than domain-specific optimizations. For example, STUN solvers [Alur et al. 2015] focus on programs with nested if-then-else operators and can leverage their structures to decompose the whole task into subtasks of synthesizing if-terms and if-conditions, thus reducing the search space exponentially with almost no cost; instead, MOLE$_{\text{FTA}}$ still needs to construct and intersect VSAs over the whole program space — its improvement in this process can hardly offset the advantage of STUN.

7 DISCUSSION AND CONCLUSION

This paper aims to accelerate VSA-based synthesis when processing many examples. Given an example set, existing VSA-based synthesizers first construct a VSA for each example and then iteratively intersect these VSAs one by one. The bottleneck here is on the second step — the iterative intersection often produces giant intermediate VSAs, leading to an unacceptable time cost.

To improve this point, we propose the scheme of *multi-way intersection*. This scheme relies on an efficient algorithm for intersecting multiple VSAs and avoids the giant intermediate VSAs by intersecting many small VSAs at once, directly into the final one. However, the existing algorithm for VSA intersection is not efficient, because it often constructs an exponential number of unnecessary nodes when the number of input VSAs increases. Hence, we further propose a novel bottom-up intersection algorithm and design a trie-based search algorithm to accelerate its procedure.

We analyze our algorithm in theory and prove its advantage compared with the previous intersection algorithm and a popular synthesis technique namely observational equivalence. On the other hand, we implement our approach into a PBE solver namely MOLE, and evaluate it over 4 different datasets and 994 synthesis tasks. The results demonstrate the advantage of MOLE against the previous VSA-based synthesizer: it solves 105 tasks more and can offer a speedup of up to ×7.36.

Combination with other PBE solvers. Though MOLE can hardly outperform some specialized PBE solvers, MOLE can be combined with these techniques to achieve further improvement. In most cases, domain properties are not enough to synthesize the whole program; instead, they can only simplify or decompose the task, and general synthesis techniques are still required to find the final result. This process leaves much room for MOLE to contribute. Below are some examples.

- Many SOTA synthesizers can be viewed as an advanced approach for constructing single-example VSAs, such as BLAZE [Wang et al. 2018], DUET [Lee 2021], FLASHFILL++ [Cambronero et al. 2023], and DRYADSYNTH [Huang et al. 2020]. MOLE can be combined with these techniques in the same paradigm as MOLE$_{\text{BLAZE}}$, which uses the existing techniques to construct single-example VSAs, and uses MOLE to perform intersection.

- Some other synthesizers use domain properties to decompose the whole task into subtasks, which are later solved by OE, such as EuSolver [Alur et al. 2017] and PolyGen [Ji et al. 2021]. In these synthesizers, Mole_{FTA} can be used to replace OE as it is more efficient.

It is future work to integrate Mole into these existing synthesizers.

Generalization to intersect other automata. Furthermore, Mole can be potentially generalized to intersect other automata. The key idea of Mole is to use multi-way intersection rather than the basic iterative intersection when the final automaton is known to be small. This prerequisite may hold in various problems; for example, many tasks require checking the emptiness of the intersection (e.g., when proving unreachability), where the final automaton is expected to be empty. On the other hand, the key idea behind our bottom-up intersection algorithm is to construct only non-empty states, which is also generalizable. It is further work to study such generalization.

ACKNOWLEDGMENTS

We thank all anonymous reviewers of this paper for their detailed and valuable suggestions. This work is sponsored 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. W2411051.

DATA-AVAILABILITY STATEMENT

{TODO: Update the link of artifact} We prepare an anonymous GitHub repository [Anonymous 2025] for the supplementary material of this paper, comprising our implementation of Mole, the dataset, all experimental results, and the appendix.

REFERENCES

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 1–8. http://ieeexplore.ieee.org/document/6679385/

Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. *CoRR* abs/1904.07146 (2019). arXiv:1904.07146 http://arxiv.org/abs/1904.07146

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I.* 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Anonymous. 2025. *Supplementary Material.* GitHub. https://github.com/molevsa/mole-artifact

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* https://openreview.net/forum?id=ByldLrqlx

Rene De La Briandais. 1959. File searching using variable length keys. In *Papers presented at the the 1959 western joint computer conference, IRE-AIEE-ACM 1959 (Western), San Francisco, California, USA, March 3-5, 1959*, R. R. Johnson (Ed.). ACM, 295–298. https://doi.org/10.1145/1457838.1457895

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL (2023), 952–981. https://doi.org/10.1145/3571226

Yuantian Ding and Xiaokang Qiu. 2024. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. *Proc. ACM Program. Lang.* 8, POPL (2024), 2129–2159. https://doi.org/10.1145/3632913

Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: web robotic process automation using interactive programming-by-demonstration. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 152–167. https://doi.org/10.1145/3519939.3523711

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

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

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1159–1174. https://doi.org/10.1145/3385412.3386027

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 215–224. https://doi.org/10.1145/1806799.1806833

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

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

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

George Karakostas, Richard J. Lipton, and Anastasios Viglas. 2003. On the complexity of intersecting finite state automata and N L versus N P. *Theor. Comput. Sci.* 302, 1-3 (2003), 257–274. https://doi.org/10.1016/S0304-3975(02)00830-7

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 776–783. http://ijcai.org/Abstract/15/115

James Koppel. 2021. Version Space Algebras are Acyclic Tree Automata. *CoRR* abs/2107.12568 (2021). arXiv:2107.12568 https://arxiv.org/abs/2107.12568

Dexter Kozen. 1977. Lower Bounds for Natural Proof Systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 254–266. https://doi.org/10.1109/SFCS.1977.16

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *CoRR* abs/1703.03539 (2017). arXiv:1703.03539 http://arxiv.org/abs/1703.03539

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434335

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

Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient Bottom-Up Synthesis for Programs with Local Variables. *Proc. ACM Program. Lang.* 8, POPL (2024), 1540–1568. https://doi.org/10.1145/3632894

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

Tom M. Mitchell. 1982. Generalization as Search. *Artif. Intell.* 18, 2 (1982), 203–226. https://doi.org/10.1016/0004-3702(82)90040-6

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *PACMPL* 2, OOPSLA (2018), 150:1–150:28. https://doi.org/10.1145/3276520

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 107–126. https://doi.org/10.1145/2814270.2814310

Oleksandr Polozov and Sumit Gulwani. 2016. Program synthesis in the industrial world: Inductive, incremental, interactive. In *5th Workshop on Synthesis (SYNT)*.

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*. 260–267. http://ijcai.org/Proceedings/75/Papers/037.pdf

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

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

Xinyu Wang. 2019. *An efficient programming-by-example framework*. Ph. D. Dissertation.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 62:1–62:26. https://doi.org/10.1145/3133886

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

Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1657–1681. https://doi.org/10.1145/3591288

A PROOFS

This appendix supplies the proofs for the lemmas and theorems in this paper.

A.1 Proof of Thm. 3.3

Theorem A.1 (Soundness of VSA Intersection). *For any set of VSAs $\mathbb{V}$, their intersection represents the set of programs shared among all individual VSAs, or formally, $\mathbb{P}\left(\bigcap_{V\in\mathbb{V}}V\right) = \bigcap_{V\in\mathbb{V}}\mathbb{P}(V)$.*

Proof. For any set of VSAs $\mathbb{V}$, their intersection represents the set of programs shared among all individual VSAs, or formally, $\mathbb{P}\left(\bigcap_{V\in\mathbb{V}}V\right) = \bigcap_{V\in\mathbb{V}}\mathbb{P}(V)$.

To get the target theorem, it is sufficient to prove that the intersection of 2 VSAs is sound. We prove this case by showing a more general result: $\mathbb{P}(\varphi(n,n')) = \mathbb{P}(n)\cap\mathbb{P}(n')$ holds for any node n in the first VSA and any node n' in the second VSA.

Our proof is conducted by induction on the total height of the sub-VSAs rooted at nodes n and n'. If the total height is 0, both n and n' must be leaf nodes. The definition of $\varphi(n,n')$ can only match case 5. By definition, we have

$$\mathbb{P}(\varphi(n,n')) = \mathbb{P}(n)\cap\mathbb{P}(n')$$

Then, for the general case, we perform case analysis on the types of n and n'.

Case 1. If both n and n' are union nodes, then they can be written as $\cup(n_1,\dots,n_k)$ and $\cup(n'_1,\dots,n'_{k'})$ respectively. The definition of $\varphi(n,n')$ matches case 1, which is

$$\varphi(n,n') = \cup(\varphi(n_i,n'_j))\ for\ i\in[1,k]\ and\ j\in[1,k'].$$

The height of nodes $n_1,\dots,n_k$ and $n'_1,\dots,n'_{k'}$ are at least 1 lower than node n and n' respectively. So, according to the induction hypothesis, for any $i\in[1,k]$ and $j\in[1,k']$:

$$\mathbb{P}(\varphi(n_i,n'_j)) = \mathbb{P}(n_i)\cap\mathbb{P}(n'_i)$$

From the definition of union nodes, for any nodes $a_1,\dots,a_m$,

$$\mathbb{P}(\cup(a_1,\dots,a_m)) = \bigcup_{i=1}^{m}\mathbb{P}(a_i)$$

Combining the equations above, we get

$$\begin{aligned}\mathbb{P}(\varphi(n,n')) &= \mathbb{P}(\cup(\varphi(n_i,n'_j)))\ for\ i\in[1,k]\ and\ j\in[1,k'] \\ &= \bigcup_{i=1}^{k}\bigcup_{j=1}^{k'}(\mathbb{P}(\varphi(n_i,n'_j))) \\ &= \bigcup_{i=1}^{k}\bigcup_{j=1}^{k'}(\mathbb{P}(n_i)\cap\mathbb{P}(n'_j)) \\ &= \bigcup_{i=1}^{k}\mathbb{P}(n_i)\cap\bigcup_{j=1}^{k'}\mathbb{P}(n'_j) \\ &= \mathbb{P}(\cup(n_1,\dots,n_k))\cap\mathbb{P}(\cup(n'_1,\dots,n'_{k'})) \\ &= \mathbb{P}(n)\cap\mathbb{P}(n')\end{aligned}$$

Case 2. If only one of n and n' is a union node, the proof is the same as case 1, as any non-union node can be regarded as a union node with one child.

Case 3. If both n and n' are join nodes of the same operator f, then they can be written as $f(n_1, \dots, n_k)$ and $f(n'_1, \dots, n'_k)$ respectively.

the definition of $\varphi(n, n')$ matches case 3, which is

$$\mathbb{P}(\varphi(n_i, n'_i)) = f(\varphi(n_i, n'_i))\ for\ i \in [1, k]$$

The height of nodes $n_1, \dots, n_k$ and $n'_1, \dots, n'_k$ are at least 1 lower than node n and n' respectively. So, according to the induction hypothesis, for any $i \in [1, k]$:

$$\mathbb{P}(\varphi(n_i, n'_i)) = \mathbb{P}(n_i) \cap \mathbb{P}(n'_i)$$

From the definition of join nodes, for any nodes $a_1, \dots, a_m$,

$$\mathbb{P}(f(a_1, \dots, a_m)) = f(\prod_{i=1}^{m} \mathbb{P}(a_i))$$

Combining the equations above, we get

$$\begin{aligned}\mathbb{P}(\varphi(n, n')) &= \mathbb{P}(f(\varphi(n_i, n'_i)))\ for\ i \in [1, k] \\ &= f(\prod_{i=1}^{k} \mathbb{P}(\varphi(n_i, n'_i))) \\ &= f(\prod_{i=1}^{k} (\mathbb{P}(n_i) \cap \mathbb{P}(n'_i))) \\ &= f(\prod_{i=1}^{k} \mathbb{P}(n_i)) \cap f(\prod_{i=1}^{k} \mathbb{P}(n'_i)) \\ &= \mathbb{P}(n) \cap \mathbb{P}(n')\end{aligned}$$

Case 4. If both n and n' are join nodes of different operators f and g respectively, the outermost operators of their corresponding programs must be f and g respectively.

As the outermost operators are different,

$$\mathbb{P}(n) \cap \mathbb{P}(n') = \emptyset$$

The definition of $\varphi(n, n')$ can only match case 4, so we have

$$\varphi(n, n') = \emptyset = \mathbb{P}(n) \cap \mathbb{P}(n')$$

Case 5. If one of n and n' is a leaf node, the definition of $\varphi(n, n')$ can only match case 5. From the definition of $\varphi(n, n')$ in case 5, we have

$$\mathbb{P}(\varphi(n, n')) = \mathbb{P}(n) \cap \mathbb{P}(n')$$

□

A.2 Proof of Lem. 3.6

Lemma A.2 (Case Study). *Consider the following grammar that involves a constant c, an input variable x, a commutative and associative operator ⊕, and a depth limit of h.*

$$S_i \coloneqq S_{i+1} \oplus S_{i+1} \quad \textbf{\textit{for}}\ i \in [1, h) \qquad\qquad S_h \coloneqq c \mid x$$

Let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the formula below provides a lower-bound for the number of nodes constructed by the top-down algorithm when intersecting $\mathbb{V}$, where $\#jnode(V, i)$ denotes the number of join nodes related to S_i.

$$\textstyle\sum_{i=1}^{h} \prod_{V \in \mathbb{V}} \#jnode(V, i)$$

If we further assume the VSA nodes distribute uniformly at each level, this lower-bound will become $(2h)^{1-|\mathbb{V}|}\prod_{V\in\mathbb{V}}|V|$*, which is close to the size of the Cartesian product.*

Proof. Consider the following grammar that involves a constant c, an input variable x, a commutative and associative operator $\oplus$, and a depth limit of h.

$$S_i \coloneqq S_{i+1}\oplus S_{i+1} \quad \textbf{for } i\in[1,h) \qquad\qquad S_h \coloneqq c \mid x$$

Let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the formula below provides a lower-bound for the number of nodes constructed by the top-down algorithm when intersecting $\mathbb{V}$, where $\#\mathsf{jnode}(V,i)$ denotes the number of join nodes related to S_i.

$$\textstyle\sum_{i=1}^{h}\prod_{V\in\mathbb{V}}\#\mathsf{jnode}(V,i)$$

If we further assume the VSA nodes distribute uniformly at each level, this lower-bound will become $(2h)^{1-|\mathbb{V}|}\prod_{V\in\mathbb{V}}|V|$, which is close to the size of the Cartesian product.

Because of the way single example VSAs are constructed, they must follow a h-level structure in this lemma. The lower part of each level is join nodes combining nodes at a lower level, and the upper part is union nodes merging join nodes corresponding to the same output.

We prove a stronger version of the first part of lemma 3.6, denoted as lemma 3.6':

When intersecting VSAs with the above structure, the top-down algorithm will construct a node for every tuple of join nodes at the same level.

We use induction on the level the node tuple is on.

On level 1, the top-down algorithm constructs the Cartesian product of all join nodes on level 1 when traversing the root node, so a node for every tuple of join nodes is constructed.

Suppose that a node for every tuple of join nodes on level $n-1$ is constructed.

We consider an arbitrary tuple of join nodes on level n $(n_1,\dots,n_k)$.

We denote the tuple of their union node fathers (or in the case that they do not have a union node father, themselves) $(f_1,\dots,f_k)$, and the fathers of $(f_1,\dots,f_k)$ $(t_1,\dots,t_k)$. $(t_1,\dots,t_k)$ are all join nodes because of the structure of individual VSAs.

f_i may be either the left child or right child of t_i. We construct another tuple of join nodes on level $n-1$ $(t'_1,\dots,t'_k)$ where f_i is always the left child of t'_i.

If f_i is the left child, $t'_i=t_i$. If f_i is the right child, because of the commutativity of $\oplus$, another join node must exist where f_i is the left child. We take this node as t'_i.

The resulting tuple $(t'_1,\dots,t'_k)$ is a tuple of join nodes on level $n-1$, so a node must be constructed in the intersection VSA.

When this node is constructed, its children is also created. In the case of $(t'_1,\dots,t'_k)$, the left child is created from the node tuple $(f_1,\dots,f_k)$.

The node tuple $(f_1,\dots,f_k)$ is a mix of union and join nodes on level n, so the intersection node is a union node whose children is the Cartesian product of the children set of each union node, and a set containing only the node itself for each non-union node. Either way, the set corresponding to node f_i always contain the node n_i, so $(n_1,\dots,n_k)$ is included in the Cartesian product, and a node is constructed for the tuple. Thus lemma 3.6' holds for every level in a VSA.

The number of tuples of join nodes at level i is $\prod_{V\in\mathbb{V}}\#\mathsf{jnode}(V,i)$, so the total number of nodes constructed is at least $\sum_{i=1}^{h}\prod_{V\in\mathbb{V}}\#\mathsf{jnode}(V,i)$.

If we further assume the VSA nodes distribute uniformly at each level, each level would have exactly $|V|/h$ nodes. Because each union node represents program sets that have the same output, each join node on each level may be the child to at most one union node, so at least half of nodes are join nodes at each level. Therefore, each level would have at least $|V|/2h$ join nodes. Replace $\#\mathsf{jnode}(V,i)$ in the first part with $|V|/2h$, and the second part of lemma 3.6 follows.

□

A.3 Proof of Thm. 3.7

Theorem A.3. *Let G be any grammar involving commutative and associative operators, and let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the following formula provides a lower-bound for the number of constructed nodes when invoking the top-down algorithm on $\mathbb{V}$.*

$$\sum_{\oplus \in G} \sum_{i>0} \prod_{V \in \mathbb{V}} \#jnode(V, \oplus, i)$$

where $\oplus$ ranges over the commutative and associative operators in G, and $\#jnode(V, \oplus, i)$ denotes the number of join nodes in V reachable from the root by passing only union nodes and exactly i $\oplus$-labeled join nodes.

Proof. Let G be any grammar involving commutative and associative operators, and let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed for a given IO example. Then, the following formula provides a lower-bound for the number of the constructed nodes when invoking the top-down algorithm on $\mathbb{V}$.

$$\sum_{\oplus \in G} \sum_{i>0} \prod_{V \in \mathbb{V}} \#\mathrm{jnode}(V, \oplus, i)$$

where $\oplus$ ranges over the commutative and associative operators in G, and $\#\mathrm{jnode}(V, \oplus, i)$ denotes the number of join nodes in V reachable from the root by passing only union nodes and exactly i $\oplus$-labeled join nodes.

We consider a subset of each V_i in $\mathbb{V}$ for each commutative and associative operator $\oplus$, denoted as $V_{i\oplus}$. The set of $V_{i\oplus}$ is denoted as $\mathbb{V}_{\oplus}$.

$V_{i\oplus}$ is defined as the VSA whose nodes are the root node and nodes in V_i that are only reachable from the root by passing only union nodes and $\oplus$-labeded join nodes, and whose edges are the edges in V_i that connect these nodes.

When we invoke the top-down algorithm on $\mathbb{V}$, the resulting VSA contains all the nodes from the VSA resulting from invoking the algorithm on $\mathbb{V}_{\oplus}$.

Following lemma 3.6, the VSA resulting from invoking the algorithm on $\mathbb{V}_{\oplus}$ contains at least $\sum_{i>0} \prod_{V \in \mathbb{V}} \#\mathrm{jnode}(V, \oplus, i)$ nodes.

For each commutative and associative operator $\oplus$ and each i, the node sets of $V_{i\oplus}$ are disjoint except for the root, so the top-down intersection of each $V_{i\oplus}$ has disjoint node sets except for the root.

Therefore, each operator $\oplus$ contributes at least $\sum_{i>0} \prod_{V \in \mathbb{V}} \#\mathrm{jnode}(V, \oplus, i)$ nodes in the complete top-down intersection.

It follows that at least $\sum_{\oplus \in G} \sum_{i>0} \prod_{V \in \mathbb{V}} \#\mathrm{jnode}(V, \oplus, i)$ nodes are in the complete top-down intersection. □

A.4 Proof of Thm. 4.4

Theorem A.4 (Soundness). *Given a set $\mathbb{V}$ of VSAs, the output of Algorithm 2 is equal to the core intersection $core(\bigcap_{V \in \mathbb{V}} V)$ if no node in $\mathbb{V}$ has both union parents and join parents.*

Proof. Given a set $\mathbb{V}$ of VSAs, the output of Algorithm 2 is equal to the core intersection $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$ if no node in $\mathbb{V}$ has both union parents and join parents.

We assume that $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$ has height h. We use V_i to denote the set of all nodes in $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$ that have height i V_i.

We prove by induction that all nodes in $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$ is constructed by the bottom-up algorithm, by proving that the set N in the bottom-up algorithm contains every node in $V_1 \cup \ldots V_i$ before the ith invocation of function buildUp.

For the case of the 1st invocation, each node in V_1 must be a leaf node, representing programs directly. If a program p is represented by some leaf node v in V_1, the program must be represented in a leaf node in each individual VSA. The bottom-up algorithm constructs a node for each different program represented by leaf nodes, so the lemma holds for this case.

Suppose that the lemma holds for V_1 through V_{i-1}. We now prove the lemma for V_i. Each node in V_i is either a join node or a union node.

Consider a join node v in V_i with operator f, constructed from the node tuple $(v^1, \dots, v^n)$. The children of v are $v_1, \dots, v_m$. The children of v^i are denoted as $v^i_1, \dots, v^i_m$ respectively.

Because v is a join node, it must be constructed by intersecting join nodes, and its ith child v_i is the intersection of the node tuple $(v^1_i, \dots, v^n_i)$. From the induction hypothesis, $v_1, \dots, v_m$ are all in N, so all the children of $v^1, \dots, v^n$ are used in some node in N.

Therefore, the node tuple $(v^1, \dots, v^n)$ is found in the buildUp function, and a node is constructed for the tuple.

Consider a union node v in V_i, constructed from the node tuple $(v^1, \dots, v^n)$, with children $v_1, \dots, v_m$.

The intersection is only necessary if node v^i has a join father or is the root of its VSA, which means node v^i has no union father. The program set of a VSA root is the program set of the entire VSA. A join node takes the Cartesian product of the program sets of each child and constructs new programs using the product. These are the only two ways a set of programs need to exist as an individual node.

Union nodes are constructed by taking the intersection of each tuple in the Cartesian product of their respective children set. Therefore, each v_i is constructed by a tuple of nodes $(v^1_i, \dots, v^n_i)$. For a fixed i, v^j_i is a child of v^j if v^j is a union node, and v^j itself if it's a join node. From the induction hypothesis, each node in $v_1, \dots, v_m$ has already been constructed by the bottom-up algorithm. Therefore, the intersections of node tuples $(v^1_i, \dots, v^n_i)$ have all been constructed in N.

We take $\varphi(v^1_1, \dots, v^n_1)$. If v^j_1 is a child of v^j, v^j must be a union father of it, so it is inserted into U_j. If $v^j_1 = v^j$, because v^j does not have a union father, v^j itself is inserted into U_j.

Therefore, we construct every node in $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$.

Note that every node is constructed from a node tuple in the Cartesian product of the node sets of each VSA. Thus, every node we construct is included in $\bigcap_{V \in \mathbb{V}} V$. Take the core set of both node sets, and our core set is a subset of $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$.

Therefore, we construct the same VSA as $\mathrm{core}(\bigcap_{V \in \mathbb{V}} V)$.

□

A.5 Proof of Lem. 4.5

Lemma A.5 (Case Study). *Consider the grammar in Lem. 3.6, repeated as follows, where c is a constant, x is the input variable, ⊕ is a commutative and associative operator, and h is a depth limit.*

$$S_i \coloneqq S_{i+1} \oplus S_{i+1} \quad \textbf{\textit{for}}\ i \in [1, h) \qquad\qquad S_h \coloneqq c \mid x$$

Let $\mathbb{V}$ be a set of VSAs on this grammar, each constructed from a given IO example. Then, Algorithm 2 will construct at most $h \cdot \mathit{size}^2(p^*)$ nodes when intersecting the VSAs in $\mathbb{V}$.

Proof. The individual VSAs follow a h-level structure. The worst case is when every program constructed from the grammar is a valid program in the intersection VSA.

Assume that level i has n_i union nodes. For level $i + 1$, the bottom-up algorithm first constructs a join node for each pair of union nodes in level i, totaling n_i^2 nodes. Then, because of the commutativity and associativity of operator ⊕, there are at most $2^i + 1$ different outputs, so only $2^i + 1$ union nodes are constructed.

Therefore, for level i, there are at most $(2^{h-2}+1)^2$ join nodes and $2^{h-1}+1$ union nodes. The size of the largest program p^* is 2^h-1. Each level has more nodes than the level below it, so the largest level is level h with $(2^{h-2}+1)^2+2^{h-1}+1$ nodes, and each level has at most $(2^{h-2}+1)^2+2^{h-1}+1$ nodes, which is less than $\text{size}^2(p^*)$ for all $h \geq 2$. Therefore, the entire VSA consisting of h layers has at most $h \cdot \text{size}^2(p^*)$ nodes.

□

A.6 Proof of Thm. 4.6

Theorem A.6 (Worst-Case Efficiency). *The following inequality always holds for any finite program space P and any example set E.*

$$cost_{Ours}(\mathbb{V}) \leq 2cost_{OE}(P,E)$$

where

- *$\mathbb{V}$ is the set of single-example VSAs constructed over program space P and example set E.*
- *$cost_{Ours}(\mathbb{V})$ denotes the number of VSA nodes constructed by our bottom-up algorithm when intersecting the VSAs in $\mathbb{V}$.*
- *$cost_{OE}(P,E)$ denotes the number of programs constructed by OE after traversing the whole program space P with the example set E.*

Proof. We consider the process in which OE constructs programs. OE maintains a set N that is the set of enumerated programs modulo observational equivalence. When a program is inserted into N, if a program with the same output behavior already exists in N, the program is discarded. For each height h and each operator f, OE enumerates all programs of height h by applying the operator f on programs in N. Then, the enumerated programs are inserted into N.

We define $N(p)$ as the program with identical output to p in n, or p itself if no program with identical output exists in N.

When our bottom-up algorithm constructs a join node of height h and operator f, it must correspond to at least one program $p = f(p_1,\dots,p_n)$. The lower nodes considered in construction each have different output behavior, so p is unique for each node constructed. Because OE traverses the entire program space, it must construct another program $p' = f(N(p_1),\dots,N(p_n))$ when enumerating programs with height h.

So, each join node corresponds to a unique program p' constructed by OE, and the number of join nodes is no larger than the number of programs constructed by OE.

Because union nodes combine join nodes with the same output behavior, one join node can have at most one union father, and the number of union nodes is no larger than the number of join nodes.

So, the total number of nodes is no larger than twice the number of join nodes, which is no larger than the number of programs constructed by OE. The theorem follows. □

B IMPLEMENTATION DETAILS

This appendix supplies the implementation details of some algorithms in this paper.

B.1 Details of Algorithm 2

Here we describe the implementation details of the construction of sets L_i, J_i and U_i.

For sets L_i (Line 6) ,before the loop (Lines 5-8) we preprocess each VSA and construct a map from each program to the set of leaf nodes comprising it. Then, the sets L_i can be taken out directly from these maps.

For sets J_i (Line 15) , for each operator f and each VSA, we maintain the set J_i as global variables and incrementally grow them as more tuples are inserted into the set N. Specifically, for each f-join node in VSA V_i, we maintain an auxiliary value denoting the number of its children that are not used in N – this value is initialized as the arity of f. Then, each time when a new tuple $\overline{n_i}$ is added to N, we visit each f-join parent of n_i, decrease its auxiliary value by 1, and add it to the corresponding J_i if the value becomes 0.

For sets U_i (Line 21), we directly enumerate the union parents of n_i. This process does not cost much time because U_i is typically small.

B.2 Details of Algorithm 3

Here we describe the details of the construction of trie for each node set N_{pre}.

Instead of constructing a new trie for each N_{pre}, we maintain the whole node set N as a trie and attach each trie node with a timestamp representing the index of the iteration (Line 9, algorithm 2) when the node is constructed. When each node is inserted into N (Line 4, algorithm 2), we set its timestamp to the index of the current iteration. During search on the trie, we ignore nodes whose timestamps are equal to the index of the current iteration, as these nodes are constructed in the current iteration and are not in the set N_{pre}.