PDF Download
3776694.pdf
10 February 2026
Total Citations: 0
Total Downloads: 102

Published: 08 January 2026
Accepted: 06 November 2025
Received: 10 July 2025

Citation in BibTeX format

RESEARCH-ARTICLE

Inductive Program Synthesis by Meta-Analysis-Guided Hole Filling

DOYOON LEE, Seoul National University, Seoul, South Korea

WOOSUK LEE, Hanyang University ERICA Campus, Ansan, Gyeonggi-do, South Korea

KWANGKEUN YI, Seoul National University, Seoul, South Korea

Inductive Program Synthesis by Meta-Analysis-Guided Hole Filling

DOYOON LEE, Seoul National University, Republic of Korea
WOOSUK LEE*, Hanyang University, Republic of Korea
KWANGKEUN YI, Seoul National University, Republic of Korea

A popular approach to inductive program synthesis is to construct a target program via top-down search, starting from an incomplete program with holes and gradually filling these holes until a solution is found. During the search, abstraction-based pruning is used to eliminate infeasible candidate programs, significantly reducing the search space. Because of this pruning, the order in which holes are filled can drastically affect search efficiency: a wise choice can prune large swaths of the search space early, while a poor choice might explore many dead-ends. However, the choice of hole-filling order is largely unattended in program synthesis literature.

In this paper, we propose a novel hole-filling strategy that leverages abstract interpretation to guide the order of hole-filling in program synthesis. Our approach overapproximates the behavior of the underlying abstract interpreter for pruning, enabling it to predict the most promising hole to fill next. We instantiate our approach to the domains of bitvectors and strings, which are commonly used in program synthesis tasks. We evaluate our approach on a set of benchmarks from the prior work, including SyGuS benchmarks, and show that it significantly outperforms the state-of-the-art approaches in terms of efficiency thanks to the abstract abstract interpretation techniques.

CCS Concepts: • **Software and its engineering** → **Programming by example**; **Automatic programming**; • **Theory of computation** → **Abstraction**; **Program analysis**.

Additional Key Words and Phrases: Program Synthesis, Programming by Example, Abstract Interpretation

ACM Reference Format:
Doyoon Lee, Woosuk Lee, and Kwangkeun Yi. 2026. Inductive Program Synthesis by Meta-Analysis-Guided Hole Filling. *Proc. ACM Program. Lang.* 10, POPL, Article 52 (January 2026), 30 pages. https://doi.org/10.1145/3776694

1 Introduction

Inductive program synthesis is a powerful technique that automatically generates programs from input-output examples, enabling the automation of various programming tasks [6, 30, 31, 35].

A popular approach to inductive program synthesis is to construct a target program via *top-down enumeration* which enumerates *partial programs* with missing parts (holes) and then gradually fills these holes to construct a complete program. Top-down enumeration is often integrated with bottom-up enumeration that generates complete subexpressions for each hole (hence *bidirectional synthesis*), allowing the synthesis engine to explore the search space more efficiently [6, 13, 23, 38].

*Corresponding author

Authors' Contact Information: Doyoon Lee, Seoul National University, Seoul, Republic of Korea, dylee@ropas.snu.ac.kr; Woosuk Lee, Hanyang University, Ansan, Republic of Korea, woosuk@hanyang.ac.kr; Kwangkeun Yi, Seoul National University, Seoul, Republic of Korea, kwang@ropas.snu.ac.kr.

ACM 2475-1421/2026/1-ART52
https://doi.org/10.1145/3776694

The power of inductive program synthesis based on top-down enumeration comes from the ability to prune the search space using abstraction-based techniques. These techniques overapproximates all possible behaviors of candidate partial programs with all possible hole fillings, allowing the synthesis engine to eliminate infeasible partial programs that cannot satisfy the specification no matter how the holes are filled.

While abstraction-based pruning is a powerful technique, an important question which has been largely overlooked in the program synthesis literature is *how to choose the order of hole-filling during the search.*

The order of hole-filling can drastically affect search efficiency: a wise choice can prune large swaths of hopeless candidate programs early, while a poor choice might explore many dead-ends. However, most existing approaches either fill holes in a fixed order (e.g., left-to-right, depth-first, or bottom-to-top) or use heuristics that do not take into account the underlying abstraction-based pruning method used in the synthesis engine. For example, consider a task of synthesizing a program that triples the gap between two integers x and y (i.e., the output is $(x - y) \times 3$) where the input-output examples are $\{x = 4, y = 1\} \mapsto 9$. Suppose we have a partial program with three holes: $(\square - \square) \times \square$, and the synthesis engine checks the feaibility of the partial program by performing the parity analysis that determines whether the partial program can produce an odd output. If we always fill the leftmost hole first, the synthesis engine will generate partial programs like $(x - \square) \times \square$ for which the parity analysis cannot determine the feasibility of the partial program, since the rightmost hole can be either odd or even. If we fill the rightmost hole first, however, the synthesis engine will generate partial programs like $(\square - \square) \times 2$ for which the parity analysis can determine that the partial program is infeasible (i.e., it cannot produce an odd output), allowing the synthesis engine to prune the search space early by discarding numerous infeasible partial and complete programs derivable from the discarded candidate.

By ignoring the possibility of good hole-filling orders, existing approaches often end up exploring large portions of the search space that could have been pruned early, or waste time on deriving inconclusive analysis results that do not help the synthesis engine to prune the search space (e.g., performing parity analysis on $(x - \square) \times \square$ wastes time).

Our Approach. Our key insight is that the order of hole-filling can be guided by analyzing the behavior of the synthesis engine's abstract interpreter, which is used to prune infeasible candidate programs during the search. From now on, we refer to this analysis as *meta analysis* and the analysis for pruning infeasible candidate programs as *underlying analysis*. We first hypothetically fill each hole in a considered partial program, and predict the result of the underlying analysis on the hypothetical candidate with the hole filled. If the meta analysis tells us that the hypothetical candidate may be determined prunable (infeasible) by the underlying analysis, we can choose the hole to fill first, so that the synthesis engine can prune the search space early. On the other hand, if the meta analysis tells us that the hypothetical candidate can never be determined prunable by the underlying analysis, we can search for other holes to fill first. If the meta analysis tells us that no matter which hole we fill first, the underlying analysis can never report prunability, we can choose any hole to fill first *without performing the underlying analysis* because the underlying analysis will not help us prune the search space anyway.

We have applied this idea to the recently proposed bidirectional synthesis algorithm Simba [38] that uses a forward-backward abstract interpreter to prune infeasible candidate programs. To this end, we have designed a meta analysis that analyzes the behavior of the underlying forward-backward abstract interpreter and an algorithm that chooses the order of hole-filling based on the result of the meta analysis. The original Simba algorithm fills holes in a fixed order (leftmost first), which is not necessarily optimal for the underlying analysis. In contrast, our algorithm fills holes in an order that is determined by the meta analysis, which can significantly reduce the search space and improve the synthesis performance. We have designed a meta analysis for the bitvector domain, which is already supported by Simba, and also a meta analysis for the string domain, which was not supported by Simba. To support the string domain, we have extended Simba by adding a new forward-backward abstract interpreter for strings.

As a result, we have implemented a new synthesis engine Hope[1] and evaluated it on a set of SyGuS benchmarks from the prior work on the domain of bitvectors and strings. Our experimental results show that Hope significantly outperforms state-of-the-art synthesis engines, including Simba, Duet [23], and SynthPhonia [13]. Also, we have conducted an ablation study to show that the meta analysis is indeed effective in guiding the hole-filling order, by showing that the synthesis performance of Hope is significantly worse when the meta analysis is disabled.

Contributions. We summarize our contributions as follows:

- A novel and general approach to inductive program synthesis that uses meta analysis to guide the order of hole-filling during the search: Unlike existing synthesis algorithms that uses a fixed order of hole-filling or heuristics that do not consider the underlying analysis, we determine hole-filling order based on the underlying abstract interpreter's behavior.
- A highly efficient meta analysis that analyzes the behavior of the underlying forward-backward abstract interpreter: We devise an efficient meta analysis that can determine whether a candidate with a hole hypothetically filled can be determined infeasible by the underlying analysis, allowing the synthesis engine to choose the order of hole-filling that can prune the search space early. Also, the meta analysis can determine whether the underlying analysis will not help prune the search space, allowing the synthesis engine to avoid performing the underlying analysis on candidates that will not be pruned.
- A highly precise and efficient abstract domain for strings: We have designed a new abstract domain for strings equipped with precise and efficient forward and backward abstract operators, which is used not only to prune infeasible candidates during the synthesis, but also as the underlying analysis for the meta analysis.
- Implementation and evaluation of our algorithm: We have implemented our algorithm in a new synthesis engine Hope and evaluated it on a set of SyGuS benchmarks from the prior work on the domain of bitvectors and strings. Our experimental results show that Hope significantly outperforms state-of-the-art synthesis engines, including Simba, Duet, and SynthPhonia.

2 Overview of Our Approach

We illustrate our method on the problem of synthesizing a string manipulating function f that takes first and last names as input (denoted x and y) and returns the initials of the names as output[2].

This problem is represented in SyGuS language, which formulates a synthesis problem as a combination of a syntactic specification and a semantic specification. The syntactic specification for f is the following grammar:

$$\begin{array}{rcll} S & \rightarrow & x \mid y \mid \texttt{"."} \mid \texttt{"␣"} & \text{string parameters and constants} \\ & \mid & \texttt{ConCat}(S, S, S) \mid \texttt{CharAt}_0(S) & \text{string operators} \end{array}$$

where S is the start non-terminal symbol, "␣" is a space character, $\texttt{ConCat}(s_1, s_2, s_3)$ concatenates three strings s_1, s_2, and s_3, and $\texttt{CharAt}_0(s)$ extracts the first character of the string s. As the semantic

[1]**H**ole-filling **O**rder **P**rediction for **E**fficient synthesis
[2]This problem is is a variant of a problem in the SyGuS benchmark suite [29].

specification, we assume only a single input-output example is given: $f("Jan", "Kotas") = "J.K"$. A solution to this problem is $f(x, y) = \mathtt{ConCat}(\mathtt{CharAt}_0(x), ".", \mathtt{CharAt}_0(y))$.

Forward-Backward Abstract Interpretation-Based Synthesis. We use a bidirectional synthesis algorithm that combines top-down and bottom-up search strategies. It first constructs a partial program with holes via top-down enumeration, and then fills the holes with components (i.e., complete subexpressions) generated via bottom-up enumeration. During this process, we use a forward-backward abstract interpreter to prune infeasible candidates. The forward abstract interpreter computes an over-approximation of the set of all possible outputs of a partial program (by considering holes placeholders for aribtrary expressions of the appropriate type), and the backward abstract interpreter computes a *necessary precondition* (i.e., a condition that must hold for the candidate to produce the desired output) for each hole.

Suppose we have a partial program with three holes [3]:

$$\mathtt{ConCat}(\square, \square, \square) \tag{1}$$

where $\square$ is a hole that can be filled with an expression of type string. Then, suppose we have a set C of components that can be used to fill the holes obtained by bottom-up enumeration. In this example, C contains the following components:

$$\{x, y, ".", "\text{␣}", \mathtt{CharAt}_0(x), \mathtt{CharAt}_0(y)\}.$$

We fill the holes in the partial program (1) with the components in C one by one. For example, suppose we fill the leftmost hole $\square$ with x obtaining the following candidate:

$$\mathtt{ConCat}(x, \square, \square) \tag{2}$$

Then, we perform the forward abstract interpretation on this candidate to compute an over-approximation of the set of all possible outputs of the candidate. We use the suffix abstraction [8] to over-approximate all possible string outputs of the candidate by representing them as the longest common suffix of the string outputs. [4] In this domain, ϵ (the empty string) is the top element meaning any string that ends with ϵ (i.e., all strings).

The forward abstract interpretation on (2) computes ϵ as the suffix abstraction of the possible outputs of the candidate. That is because the other holes that can be filled with any string expressions will affect the suffix representation of the possible outputs, but they are yet to be filled. Then, we check if this abstraction *subsumes* the desired output $"J.K"$ (i.e., the desired output is a possible output of the candidate). If not, the candidate is determined infeasible and discarded. In this case, ϵ does subsume $"J.K"$, and thus the candidate (2) is determined feasible.

Please note that the candidate (2) is actually infeasible, because it can never produce the desired output $"J.K"$ (the output will always be of the form $"Jan"$ followed by some string). However, the imprecision of the suffix abstraction leads to the false positive.

Next, we proceed to perform backward abstract interpretation to compute necessary preconditions for the other holes to be filled. Because the desired output ends with $"K"$, the backward abstract interpreter computes the necessary precondition $"K"$ for the rightmost hole, which means that the hole must be filled with an expression that evaluates to a string ends with $"K"$.

This necessary precondition is used to identify the components that can be used to fill the last hole. For example, the component $"."$ cannot be used to fill the rightmost hole but $\mathtt{CharAt}_0(y)$ can be used.

[3] We exhaustively enumerate all possible partial programs by applying the production rules of the grammar to the start non-terminal up to a certain depth. In this example, we assume we only generates a single partial program for simplicity.
[4] In the actual implementation, we use a more precise abstract domain for strings, but we use the suffix abstraction here for simplicity.

By these bidirectional abstract interpretations, the search space is pruned by

- discarding infeasible candidates that cannot produce the desired output, and
- discarding components that cannot be used to fill the holes.

Hole Filling Orders in Existing Approaches. The most popular approach to hole filling is to fill the holes in a fixed order. The leftmost-first order is the most common choice. However, this order is not always optimal for the underlying abstract interpreter. For example, if we fill the leftmost hole first, we generate the candidate (2) and the forward abstract interpretation on it cannot determine infeasibility because of the other remaining holes. Then, we will proceed to fill the other holes assuming the candidate can be feasible, wasting time on infeasible candidates.

In contrast, if we choose the rightmost hole, for example, we may generate the candidate

$$\texttt{ConCat}(\square, \square, \texttt{"␣"}). \tag{3}$$

Then, the forward abstract interpretation computes "␣" as the suffix abstraction of the possible outputs of the candidate, realizes that the candidate can never produce the desired output "$J.K$", and determines it infeasible. This example shows that the order of hole filling should be chosen *being aware of the underlying abstract interpreter* to prune the search space effectively.

Meta Analysis. Our key insight is that the order of hole filling can be guided by predicting the behavior of the underlying abstract interpreter.

To overapproximate all possible behaviors of the underlying abstract interpreter, we introduce a simple abstract domain $D^\# = \{\texttt{Must}\top, \texttt{May}\not\top\}$ where $\texttt{Must}\top$ means that the underlying abstract interpreter will always return $\top$ (ϵ in the suffix abstraction) on a candidate, and $\texttt{May}\not\top$ means that the underlying abstract interpreter may not return $\top$ on a candidate (but it may return $\top$ as well). This domain is simple enough to make our meta analysis lightweight and efficient, which is crucial for our approach because the cost of the meta analysis should not outweigh the cost of the underlying abstract interpreter. [5]

The domain $D^\#$ has forward and backward abstract operators that soundly overapproximate the behavior of their underlying counterparts. For example, the forward abstract interpretation with $D^\#$ computes $\texttt{Must}\top$ for $\texttt{ConCat}(x, \square, \square)$ since the underlying forward abstract interpreter will always return $\top$ on this candidate, and the backward abstract interpretation with $D^\#$ computes $\texttt{May}\not\top$ for the necessary precondition of the hole $\square$ in $\texttt{ConCat}(\texttt{CharAt}_0(x), \texttt{"."}, \square)$ since the underlying backward abstract interpreter can infer the suffix abstraction of $\square$ as "K" to produce the desired output "$J.K$".

Meta-Analysis-Guided Hole Filling. Now we explain how to use the meta analysis to guide the order of hole filling. Given the partial program (1), we want to know which hole to fill first. To this end, we hypothetically fill each hole and generate three candidates:

$$\texttt{ConCat}(\boxdot, \square, \square) \tag{4}$$

$$\texttt{ConCat}(\square, \boxdot, \square) \tag{5}$$

$$\texttt{ConCat}(\square, \square, \boxdot) \tag{6}$$

where $\boxdot$ represents a hole that is hypothetically filled with some component. By "hypothetically filled", we mean that we do not actually fill the hole, but rather analyze the underlying abstract interpreter's behavior assuming the hole is filled with an arbitrary component. The abstract semantics of $\boxdot$ in the meta analysis is $\texttt{May}\not\top$ because the $\boxdot$ symbolizes a complete expression

[5] In our implementation, we use this simple domain $D^\#$ for bit-vectors, but we use $D^\# \times \mathbb{I}^\#$ for suffix abstraction of strings where $\mathbb{I}^\# = \{\texttt{Con}, \texttt{Bnd}, \texttt{Unb}\}$ represents an abstraction of string lengths: constant length, bounded length, and unbounded length. In this example, we use the simple domain $D^\#$ for ease of explanation.

without any holes in it, and the underlying abstract interpreter will never return $\top$ for it as there is no unknown part in it.

Then, we perform the forward meta analysis on the above candidates (4)–(6). The following process summarizes the meta analysis on the candidates:

- For candidate (4), the forward meta analysis computes $\texttt{Must}\top$ as the suffix abstraction of the candidate because the last hole can be filled with any string expression, which makes the suffix abstraction $\top$.
- For candidate (5), the forward meta analysis computes $\texttt{Must}\top$ for a similar reason as the first candidate.
- For candidate (6), the forward meta analysis computes $\texttt{May}\not\top$ as the suffix abstraction of the candidate because the last hole $\boxdot$ will determine the suffix abstraction of the candidate, and it will not evaluate to $\top$ in the underlying analysis because a complete expression will evaluate to a non-empty string.

Next, we perform the backward meta analysis on the candidates (4)–(6). The backward meta analysis on (4) computes $\texttt{May}\not\top$ as the necessary precondition for the entire candidate because it should evaluate to the desired output "$J.K$" whose suffix is definitely non-empty. Then, it proceeds to compute the necessary precondition for the other sub-expressions (including the other holes) in the candidate. For example, the necessary precondition for the rightmost hole is $\texttt{Must}\top$. That is because we need information about the middle hole to determine the necessary precondition for the rightmost hole, and the middle hole is not determined yet. In this manner, the backward meta analysis computes the necessary preconditions for all the sub-expressions of the candidates (4)–(6).

After the forward and backward meta analyses, we obtain the forward analysis result and the backward analysis result for each subexpression in each candidate. We refer to each subexpression as a position of it: ϵ denotes the root position of the candidate, and the positions 1, 2, and 3 denote the first, second, and third subexpressions of the root position, respectively. The following table summarizes the results.

Candidate	Position	Results of the Forward & Backward Meta Analyses
(4)	ϵ	$(\texttt{Must}\top, \texttt{May}\not\top)$
	1	$(\texttt{May}\not\top, \texttt{Must}\top)$
	2	$(\texttt{Must}\top, \texttt{Must}\top)$
	3	$(\texttt{Must}\top, \texttt{Must}\top)$
(5)	ϵ	$(\texttt{Must}\top, \texttt{May}\not\top)$
	1	$(\texttt{Must}\top, \texttt{May}\not\top)$
	2	$(\texttt{May}\not\top, \texttt{Must}\top)$
	3	$(\texttt{Must}\top, \texttt{Must}\top)$
(6)	ϵ	$(\texttt{May}\not\top, \texttt{May}\not\top)$
	1	$(\texttt{Must}\top, \texttt{Must}\top)$
	2	$(\texttt{Must}\top, \texttt{Must}\top)$
	3	$(\texttt{May}\not\top, \texttt{Must}\top)$

Given the above results, now we are ready to determine the order of hole filling. We can see that the candidate (6) may lead to the best pruning effect. Because the meta forward analysis result of the root position is $\texttt{May}\not\top$, it means that the underlying abstract interpreter may not return $\top$ on the candidate. The meta backward analysis result of the root position is also $\texttt{May}\not\top$, which means there may be a non-trivial necessary precondition for the root position. Recall the candidate 3. Its underlying forward analysis result is "␣" (non-$\top$), and the desired output is "$J.K$" (also non-$\top$). There is no intersection between the two, which means that the candidate can never produce the desired output "$J.K$". The meta analysis result of the root position indicates the possibility of such a situation, choosing the rightmost hole is the best choice to fill first.

If we do not choose the rightmost hole to fill first, filling the middle hole first is the second best choice. The reason is as follows: though the forward analysis result of the root position is `Must`$\top$ (i.e., the candidate cannot be pruned by the intersection of the suffix abstraction and the desired output), the backward analysis result of the rightmost hole is `May`$\not\top$. We may infer a non-trivial necessary precondition for the rightmost hole, which can lead to pruning the search space by limiting the components that can be used to fill the rightmost hole.

Choosing the leftmost hole to fill first is the worst choice because the forward analysis result of the root position is `Must`$\top$, and the backward analysis results of the other remaining holes are both `Must`$\top$. This means that not only we cannot prune the candidate but also we cannot infer any non-trivial necessary precondition for the other holes.

We use the following ranking hierarchy to determine the order of hole filling:

- **Rank-1 positions**: The positions with hypothetical filling that lead to `May`$\not\top$ as the forward analysis result and `May`$\not\top$ as the backward analysis result at any position. Filling these positions first is the best choice because it may lead to the best pruning effect by discarding the entire candidate and all candidates that can be derived from it.
- **Rank-2 positions**: The positions with hypothetical filling that leads to `Must`$\top$ as the forward analysis result and `May`$\not\top$ as the backward analysis result at any other hole's position. Filling these positions first is the second best choice because it may lead to a non-trivial necessary precondition for the other holes, so that the search space can be pruned by limiting the components that can be used to fill the other holes.
- **Rank-3 positions**: The positions that are neither Rank-1 nor Rank-2. Filling these positions first is the worst choice because it cannot lead to any pruning effect, and the search space will be explored in a less efficient way.

In our example, the rightmost hole is a Rank-1 position, the middle hole is a Rank-2 position, and the leftmost hole is a Rank-3 position.

If there are no Rank-1 and Rank-2 positions, we can choose any Rank-3 position to fill first and importantly, we do not need to perform the underlying analysis after filling the hole because it will not help prune the search space anyway. This way, we avoid fruitless analysis on candidates that will not be pruned (for more details of the meta analysis and ranking process for this example, please refer to Examples 4.1 and 4.3 in §4).

3 Problem Definition

We define the problem of finding a good hole-filling strategy in example-based program synthesis.

3.1 Preliminaries

Regular Tree Grammar. A regular tree grammar is a tuple $G = (\mathcal{N}, \Sigma, S, \delta)$ where $\mathcal{N}$ is a finite set of nonterminal symbols; Σ is a ranked alphabet; $S \in \mathcal{N}$ is the start nonterminal; and δ is a finite set of productions of the form $N_0 \to \sigma^{(i)}(N_1, \cdots, N_i)$ where each $N_j \in \mathcal{N}$ is a nonterminal and $\sigma^{(i)}$ is a k-ary function symbol in Σ.

A partial program P is a tree possibly containing nonterminals, and each subtree of P can be identified with a position in P (denoted by $Pos(P)$). A position in a partial program P is a string over the alphabet of natural numbers where the empty string ϵ denotes the root position of P and for a position p, the position $p \cdot j$ denotes the j-th child of the node at position p (where $\cdot$ is the concatenation operator). We will denote by $P \mid_p$ the subprogram of P at position p and by $P[p \leftarrow t]$ the program that is obtained from P by replacing the subprogram at position p with t. Given a partial program P, we denote by $\mathsf{Holes}(P)$ the set of all positions in P where a nonterminal appears, i.e., $\mathsf{Holes}(P) = \{p \in Pos(P) \mid P\mid_p \in \mathcal{N}\}$. We will use the term hole to refer to a nonterminal in a partial program. A program P' can be derived in one step from P if there exists a production rule $N_0 \rightarrow \sigma^{(i)}(N_1, \cdots, N_i)$ such that replacing N_0 in P with $\sigma^{(i)}(N_1, \cdots, N_i)$ – denoted by $P \rightarrow P'$. We say that a partial program P' is derived from another partial program P in zero or more steps (denoted by $P \rightarrow^* P'$) if P' can be obtained from P by applying a sequence of productions.

Given an RTG G, we say a partial program P is *complete* if it does not contain any holes, i.e., $\mathsf{Holes}(P) = \emptyset$. We use $\mathcal{L}(N)$ to denote the set of all complete programs that can be derived from a nonterminal N in G, hence $\mathcal{L}(G) = \mathcal{L}(S)$ is the set of all complete programs that can be derived from the start nonterminal S in G.

Example-based Syntax-Guided Synthesis. An example-based syntax-guided synthesis (SyGuS) instance is a tuple $\langle G, \mathcal{E} \rangle$ where G is a regular tree grammar and $\mathcal{E}$ is a set of input-output examples. The goal is to find a program P satisfying a given specification $\mathcal{E}$. Programs must be written in a language $L(G)$ described by a regular-tree grammar G. The specification is a set of input-output pairs $\mathcal{E} = \bigcup_{j=1}^{m} \{i_j \mapsto o_j\}$ where i_j and o_j are values. Assuming each program P in $L(G)$ has a deterministic semantics $[\![P]\!]$, the goal is to find a program P such that $[\![P]\!]\,(i) = o$ for all $i \mapsto o \in \mathcal{E}$ (denoted $P \models \mathcal{E}$).

3.2 Synthesis with Abstract Interpretation-based Pruning

We consider a generic synthesis algorithm that uses abstract interpretation to prune the search space. Algorithm 1 describes the algorithm that takes as input an example-based SyGuS instance $\langle G, \mathcal{E} \rangle$ and an abstract interpreter $\mathscr{A}$. The algorithm maintains a queue Q of partial programs, initialized with sketches generated by a sketch generator InitialSketches(G) (line 2). The algorithm also maintains a component map $\mathbf{C}$ that maps each nonterminal N to a set of components that can be used to fill holes in N (line 3). The algorithm then repeatedly explores the queue Q (lines 6 – 18). In each iteration, the algorithm generates new components for each nonterminal N in the grammar G (line 5). Then, it processes each partial program P in the queue Q (line 6). For each P, it first removes P from the queue (line 7). If P is complete and satisfies the specification $\mathcal{E}$, the algorithm returns P as a solution (line 8). Otherwise, it analyzes P using the abstract interpreter $\mathscr{A}$ (line 10). If the analysis determines that P is infeasible with respect to the specification $\mathcal{E}$, the algorithm continues to the next iteration (line 11). If P is feasible, the algorithm picks a position *pos* in P where a hole exists (line 12). For each component c in $\mathbf{C}(P\mid_{pos})$ that is *compatible* with the analysis result $\mathscr{A}$ (i.e., it does not violate any *necessary precondition* over the hole imposed by the analysis), the algorithm enqueues a new partial program $P[pos \leftarrow c]$ into the queue Q (lines 14–15). If the queue Q becomes empty, the algorithm goes back to line 5 to generate new additional components for the next iteration to broaden the search space. The entire algorithm continues this process until the budget is exhausted (line 19).

This generic algorithm is general enough to cover various synthesis techniques, including top-down synthesis, bidirectional synthesis, and unrealizability proof. For example, the abstract interpreter $\mathscr{A}$ can be either a forward abstract interpreter [16, 20, 32–37, 37] or a forward-backward abstract interpreter [25, 38]. If only a forward interpreter is used, the condition on line 14 is always true, since there are no necessary preconditions over holes imposed by the analysis. If the AddComponents function is a constant function that returns a set of production rules for each nonterminal N, the algorithm will behave like a standard top-down synthesis algorithm. If the AddComponents function generates complete subexpressions for each nonterminal N, the algorithm will behave like a bidirectional synthesis algorithm [23, 38]. Lastly, if all the partial programs in the queue are determined infeasible by the analysis, the algorithm can prove the unrealizability

Algorithm 1 Abstract Interpretation-based Inductive Synthesis

```
Require: A SyGuS instance $\langle G, \mathcal{E} = \bigcup_{j=1}^{m} \{i_j \mapsto o_j\} \rangle$
Require: Abstract interpreter ANALYZE
Ensure: A solution program $P \in L(G)$ that satisfies $\mathcal{E}$
 1: procedure SYNTHESIS($\langle G, \mathcal{E} \rangle$, ANALYZE)
 2:     $Q$ := INITIALSKETCHES($G$)
 3:     $\mathbf{C} := \emptyset$
 4:     repeat
 5:         $\mathbf{C}$ := ADDCOMPONENTS($G, \mathbf{C}$)                              ▷ $\mathbf{C} : N \to \mathcal{P}(L(G))$
 6:         while $Q$ is not empty do
 7:             remove $P$ from $Q$
 8:             if IsComplete($P$) and $P \models \mathcal{E}$ then return $P$
 9:             else
10:                 $\mathcal{A}$ := ANALYZE($P, \mathcal{E}$)
11:                 if Infeasible($P, \mathcal{A}, \mathcal{E}$) then continue
12:                 $pos$ := PickHole($P$)                                   ▷ $pos \in Pos(P)$
13:
14:                 for $c \in \mathbf{C}(P\,|_{pos})$ s.t. $\neg$Infeasible($P[pos \leftarrow c], \mathcal{A}, \mathcal{E}$) do
15:                     $Q := Q \cup \{P[pos \leftarrow c]\}$
16:                 end for
17:             end if
18:         end while
19:     until the budget is exhausted
20: end procedure
```

of the given SyGuS problem [18, 19] assuming the soundness of the abstract interpreter $\mathcal{A}$ and the completeness of the sketch generator INITIALSKETCHES(G).

Throughout this paper, we will assume that the abstract interpreter $\mathcal{A}$ is sound with respect to the underlying semantics of the programs in $L(G)$, i.e., if a program is determined infeasible by the analysis, then it is indeed infeasible with respect to the specification $\mathcal{E}$.

3.3 Problem Statement

We first define the notion of hole-filling strategies and then introduce our problem of finding an admissible hole-filling strategy.

Definition 3.1. For a given program $P \in L(G)$, a hole-filling strategy is a function PickHole : $L(G) \to Pos(P)$ that takes the partial program P and returns a position $pos \in Pos(P)$ where a hole exists, i.e., $P\,|_{pos} \in \mathcal{N}$. A hole-filling strategy may be *non-deterministic* if it randomly chooses one position from multiple hole positions that are considered equally promising by the strategy.

Different hole-filling strategies can be used in Algo. 1 on line 12 to determine which hole to fill next in a partial program P. The choice of a hole-filling strategy can significantly affect the search space and the efficiency of the synthesis algorithm. A typical example of hole-filling strategies is the one that always picks the leftmost hole in P. However, this strategy may not always be the best choice, as it may not lead to the most effective search space pruning.

A hole-filling strategy determines how the holes in a partial program P are filled with components from a current component map $\mathbf{C}$. We define the notion of a hole expansion set as follows:

Definition 3.2. A hole expansion set of a partial program P with respect to a component map $\mathbf{C}$ and a position *pos* of a hole to be filled is defined to be

$$\mathsf{HoleExpansion}(P, pos, \mathbf{C}) = \{P[pos \leftarrow c] \mid c \in \mathbf{C}(P\mid_{pos})\}$$

The hole expansion set $\mathsf{HoleExpansion}(P, pos, \mathbf{C})$ contains all the partial programs that can be obtained from P by filling a hole at position *pos* with a component c from the component map $\mathbf{C}$.

If a hole is filled in a way that necessary preconditions over the other holes are constraining enough to prune the search space, we say that the hole-filling strategy is effective because our primary goal is to prune the search space as much as possible. This notion of effectiveness can be quantified by the pruning power of a hole-filling strategy, which is defined as follows:

Definition 3.3. Given a partial program $P \in L(G)$, a component map $\mathbf{C}$, and a synthesis specification $\mathcal{E}$, the pruning power of a hole-filling strategy PickHole is defined to be

$$\Theta(P, \mathsf{PickHole}, \mathbf{C}) = \max_{pos \in [\![\mathsf{PickHole}]\!](P)} 1 - \frac{|\{p \in \mathsf{HoleExpansion}(P, pos, \mathbf{C}) \mid \exists p' \in L(G).\, p \rightarrow^* p' \wedge p' \models \mathcal{E}\}|}{|\mathsf{HoleExpansion}(P, pos, \mathbf{C})|}$$

where $[\![\mathsf{PickHole}]\!]\,(P)$ is the set of all positions that may be returned by the (possibly non-deterministic) hole-filling strategy PickHole on input P, $p \rightarrow^* p'$ means that p' can be derived from p in zero or more steps, and $p' \models \mathcal{E}$ means that p' satisfies the specification $\mathcal{E}$.

The best hole-filling strategy is the one that maximizes the Θ value. Ideally, if a hole-filling strategy has a pruning power of 1, it means that all the programs in the hole expansion set are infeasible with respect to the specification $\mathcal{E}$, hence we can discard the current partial program P and avoid further exploration of the search space from P. On the other hand, if a hole-filling strategy has a pruning power of 0, it means that all the programs in the hole expansion set are determined to be feasible with respect to the specification $\mathcal{E}$ by the analyzer hence we cannot prune any candidate program.

However, checking the feasibility of a partial program P with respect to the specification $\mathcal{E}$ in a sound and complete manner is undecidable in general. Therefore, we use the following approximation:

Definition 3.4. Given an abstract interpreter ANALYZE, the estimated pruning power of a hole-filling strategy PickHole is defined to be

$$\widehat{\Theta}(P, \mathsf{PickHole}, \mathbf{C}) = \max_{pos \in [\![\mathsf{PickHole}]\!](P)} 1 - \frac{|\{p \in \mathsf{HoleExpansion}(P, pos, \mathbf{C}) \mid \neg\mathsf{Infeasible}(p, \mathcal{A}, \mathcal{E})\}|}{|\mathsf{HoleExpansion}(P, pos, \mathbf{C})|}$$

where $\mathcal{A} = \textsc{Analyze}(P, \mathcal{E})$ is the analysis result of the abstract interpreter ANALYZE.

Note that we use the ANALYZE function to determine if a partial program P is infeasible with respect to the specification $\mathcal{E}$.

Definition 3.5. A hole-filling strategy $\mathsf{PickHole}^*$ with respect to a partial program $P \in L(G)$ and a current component map $\mathbf{C}$ is *optimal* if it maximizes the pruning power, i.e.,

$$\forall \mathsf{PickHole}' \in L(G) \rightarrow Pos(P).\ \widehat{\Theta}(P, \mathsf{PickHole}^*, \mathbf{C}) \geq \widehat{\Theta}(P, \mathsf{PickHole}', \mathbf{C})$$

The problem of finding an optimal hole-filling strategy can be stated as follows:

Definition 3.6. At every iteration of the synthesis algorithm in Algo. 1 on line 12, given a partial program $P \in L(G)$ and a current component map $\mathbf{C}$, the problem of finding an optimal hole-filling strategy is to find a hole-filling strategy $\mathsf{PickHole}^*$ of the best pruning power, i.e.,

$$\mathsf{PickHole}^* = \operatorname*{arg\,max}_{\mathsf{PickHole} \in L(G) \rightarrow Pos(P)} \widehat{\Theta}(P, \mathsf{PickHole}, \mathbf{C}).$$

A naive approach to finding an optimal hole-filling strategy is to enumerate all possible hole-filling strategies and compute their estimated pruning power. This approach is infeasible in practice because it requires to run the abstract interpreter ANALYZE for every candidate program in the hole expansion set of every possible hole-filling strategy.

Finding a hole-filling strategy should be efficient enough not to slow down the synthesis algorithm, and thus we cannot afford to compute the pruning power of every possible hole-filling strategy.

Therefore, we relax the problem of finding an optimal hole-filling strategy to finding a hole-filling strategy that can recognize the worst-case scenario where no pruning can be achieved regardless of how holes are filled. We call such a hole-filling strategy *admissible*.

Definition 3.7. For a partial program $P \in L(G)$, a component map $\mathbf{C}$, a hole-filling strategy PickHole is *admissible* if it satisfies the following condition:

$$\widehat{\Theta}(P, \mathsf{PickHole}, \mathbf{C}) = 0 \implies \forall \mathsf{PickHole}' \in L(G) \to Pos(P).\ \widehat{\Theta}(P, \mathsf{PickHole}', \mathbf{C}) = 0$$

If the admissible strategy shows the worst pruning power (i.e., 0), then no other strategy can achieve better. As a contrapositive, if there exists a hole-filling strategy PickHole′ whose pruning power is not the worst (i.e., greater than 0), then the admissible hole-filling strategy PickHole is guaranteed to have a pruning power strictly better than the worst case.

Finding an admissible hole-filling strategy is beneficial: when its pruning power is 0, the search space cannot be pruned no matter how holes are filled. Therefore, we can safely continue the synthesis process without performing the analysis on the hole expansion set to prune the search space. This can significantly reduce the number of calls to the abstract interpreter ANALYZE and speed up the synthesis process.

To summarize, we require the hole-filling strategy to be admissible and to check if the pruning power of the hole-filling strategy is 0. The lines 12–13 in Algorithm 1 are modified as follows:

```
...                                                          ▷ Lines before 12 are unchanged
$pos, noprune := \mathsf{PickHole}(P)$                        ▷ $pos \in Pos(P)$
**if** $noprune$ **then** $Q := Q \cup \{P[pos \leftarrow c] \mid c \in \mathbf{C}(P\,|_{pos})\}$;  **continue**
...                                                          ▷ Lines from 14 are unchanged
```

The variable *noprune* is a boolean value that indicates whether the pruning power is 0 or not. If *noprune* is true, it means that the pruning power of the hole-filling strategy is 0, and thus we can safely continue the synthesis process without performing the analysis on the hole expansion set to find infeasible combinations of holes and components.

In the following sections, we will present a general method for obtaining an admissible hole-filling strategy.

4 Our Meta Analysis-based Hole-Filling Strategy

In this section, we present our abstract interpretation-based hole-filling strategy. Our method can be instantiated to various domains of interest such as bitvector and string manipulation by providing forward and backward abstract operators for our meta abstract domain.

We first present a top-level algorithm that uses a meta abstract interpreter to analyze the behavior of the underlying abstract interpreter for pruning. Then, we describe the underlying abstract interpreter and the meta abstract interpreter in detail.

4.1 Top-Level Algorithm

Our top-level algorithm for hole-filling is presented in Algo. 2. The procedure takes a partial program P with holes and a meta abstract interpreter ANALYZE$^{\#}$ as inputs, and returns a set of positions to be filled (note that it takes $\text{Analyze}^{\#}$ in addition to P in contrast to the one in Algo. 1 that takes only P). The procedure initializes two sets of positions on lines 2 and 3: (1) **Rank1_positions**: filling a hole in the position with some component may lead to a candidate that is infeasible with respect to the input-output example, and (2) **Rank2_positions**: filling a hole in the position with some component may lead to a candidate in which the holes have non-trivial (i.e., not $\top$) necessary preconditions, and thus the components that can be used to fill the holes are pruned. If the algorithm identifies a Rank-1 position, it returns the position because filling the hole in the position with any component will lead to an infeasible candidate, which is the best choice for pruning. If the algorithm identifies a Rank-2 position, it returns the position because filling the hole in the position with any component will lead to a candidate with necessary preconditions over holes, which is also a good choice for pruning. If neither Rank-1 nor Rank-2 positions are found, it randomly picks a position to be filled and returns it with a boolean value indicating that the pruning power is 0. After the initialization, the algorithm iterates over all positions in the program P with holes (line 4). For each position *pos* in the program, it performs the following steps:

- It analyzes the program P with a *hypothetical* filling of the position *pos* using the meta abstract interpreter $\text{Analyze}^{\#}$ (line 5). By hypothetical filling, we mean that the position is not actually filled with any component, but the analysis is performed as if it were filled with some component.
- It checks if the position may lead to an infeasible candidate (line 6). If so, it adds the position to the Rank-1 positions set (line 7).
- If not, it checks if the position may prune components (line 8) by enabling the necessary preconditions over holes. If so, it adds the position to the Rank-2 positions set (line 9).
- If neither condition is satisfied, it continues to the next position.

If the algorithm finds any Rank-1 positions, it returns one of them (line 13) with a boolean value indicating that the pruning power may not be 0. If it finds any Rank-2 positions, it returns one of them (line 15) with a boolean value indicating that the pruning power may not be 0. If it does not find any Rank-1 or Rank-2 positions, it randomly picks a position to be filled (line 17) and returns it with a boolean value indicating that the pruning power must be 0.

The core component of the algorithm is the meta abstract interpreter $\text{Analyze}^{\#}$ that analyzes the behavior of the underlying abstract interpreter Analyze over the program P with a hypothetical filling of a given position.

To describe the meta abstract interpreter in detail, we need to first describe the underlying abstract interpreter and the component generation process we suppose in this paper. We assume that the underlying abstract interpreter Analyze is a forward-backward abstract interpreter that computes both the overapproximation of the output of a given partial program and the necessary preconditions over missing holes for the program to be valid. The component generation process is a procedure that generates complete subexpressions without holes according to the grammar of the target language. It may be implemented in various ways (e.g., bottom-up enumeration).

4.2 Underlying Analysis Analyze

We now formally describe the Analyze procedure in Algo. 1 we assume in this paper. The following description is borrowed from the work of Simba [38]. The Analyze procedure is based on alternating forward and backward analyses that can compute an overapproximation of the set of states that are both reachable from the program entry and able to reach a desired state at the program exit [9]. In our setting where a program is a tree, execution of a program begins at the leaves of the program tree (constants or the input parameter variable) and proceeds up to the root. Therefore, for each node of the program tree, a forward analysis computes an over-approximation of the set of values

Algorithm 2 Meta Analysis Analyze$^\#$-Based Hole-Filling

```
Require: A partial program $P$ with holes
Require: Meta abstract interpreter Analyze$^\#$
Ensure: A position to be filled in $P$ and a boolean value indicating whether the pruning power is 0
1: procedure PickHole($P$, Analyze$^\#$)
2:     Rank1_positions := $\emptyset$
3:     Rank2_positions := $\emptyset$
4:     for $pos \in$ Holes($P$) do
5:         $\mathcal{A}^{\text{Meta}}$ := Analyze$^\#$($P, pos$)
6:         if MayInfeasible ($P, \mathcal{A}^{\text{Meta}}$) then
7:             Rank1_positions := Rank1_positions $\cup \{pos\}$
8:         else if MayPruneComponents ($P, \mathcal{A}^{\text{Meta}}$) then
9:             Rank2_positions := Rank2_positions $\cup \{pos\}$
10:        end if
11:    end for
12:    if Rank1_positions $\neq \emptyset$ then
13:        return Pick(Rank1_positions), false
14:    else if Rank2_positions $\neq \emptyset$ then
15:        return Pick(Rank2_positions), false
16:    else
17:        return Pick(Holes($P$)), true
18:    end if
19: end procedure
```

that may be computed from a subtree rooted at the node in a bottom-up manner. A backward analysis computes an over-approximation of the set of values that may be used to generate output desired by its parent in a top-down fashion.

For brevity, we assume that only a single input-output example $i \mapsto o$ is given to the Analyze procedure, but it can be easily extended to multiple input-output examples by performing the analysis for each example and combining the results.

We suppose an underlying abstract domain $\hat{D}$ has a galois connection with the concrete domain D (i.e., $\mathcal{P}(D) \underset{\alpha_{\hat{D}}}{\overset{\gamma_{\hat{D}}}{\leftrightarrows}} \hat{D}$). The Analyze procedure performs the iterative forward-backward analysis to obtain a map $\mathcal{A}$ that maps each position in P to an abstract values $\hat{d}$.

The forward abstract semantics is characterized by the least fixpoint of the following function $\mathcal{F} : (Pos(P) \to \hat{D}) \to (Pos(P) \to \hat{D})$:[6]

$$\mathcal{F} = \lambda X.\, \mathcal{I}_{\mathcal{F}} \sqcup F^\#(X) \quad \text{where} \quad \mathcal{I}_{\mathcal{F}} = \{p \mapsto \begin{cases} \alpha_{\hat{D}}(\{P\,|_p\}) & (P\,|_p \text{ is a constant}) \\ \bot & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}.$$

Each constant is mapped to the abstraction of the constant itself, and nonterminals are mapped to $\top$ by $F^\#$. The forward abstract function $F^\#$ is defined as follows:

$$F^\#(X) = \{p \mapsto \begin{cases} \top & (P\,|_p \in N) \\ \overrightarrow{f}^\#(X(p\cdot 1), \cdots, X(p \cdot k)) & (P\,|_p = f(\cdots), arity(f) = k) \\ \bot & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}$$

[6]For simplicity, we assume that every occurrence of the input parameter variable is replaced with the input example value i, which is a constant.

where $\overrightarrow{f}^{\#}$ denotes the forward abstract operator of an operator f. It takes abstract values of arguments and returns a resulting abstract value.

The backward abstract semantics is characterized by the greatest fixpoint of the following function $\mathcal{B} : (Pos(P) \to \hat{D}) \to (Pos(P) \to \hat{D})$:

$$\mathcal{B} = \lambda X.\, \mathcal{I}_{\mathcal{B}} \sqcap B^{\#}(X) \quad \text{where} \quad \mathcal{I}_{\mathcal{B}} = \{p \mapsto \begin{cases} \alpha_{\hat{D}}(\{o\}) & (p = \epsilon) \\ \top & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}.$$

The final state $\mathcal{I}_{\mathcal{B}}$ maps the root position to the abstraction of the output example (i.e., $\alpha_{\hat{D}}(\{o\})$) and every other position to $\top$. The backward abstract function $B^{\#}$ is defined as follows:

$$B^{\#}(X) = \{p \mapsto \begin{cases} \overleftarrow{f}^{\#}_{i}(X(p'), X(p' \cdot 1), \cdots, X(p' \cdot k)) & (p = p' \cdot i, P\mid_{p'} = f(\cdots), arity(f) = k) \\ \top & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}$$

where $\overleftarrow{f}^{\#}_{i}$ denotes the backward abstract operator of an operator f. It takes an abstract value of the result and abstract values of arguments, and returns an abstract value of the i-th argument, which corresponds to the *necessary precondition* for the i-th argument to satisfy the input-output example.

The intersection of the forward and backward abstract semantics is computed by obtaining the limit of the following decreasing chain defined for all $n \in \mathbb{N}$ until the chain converges [9]: $\dot{X}^0 = lfp\, \mathcal{F}$, $\dot{X}^{2n+1} = gfp\, \lambda X.\, \dot{X}^{2n} \sqcap \mathcal{B}(X)$, and $\dot{X}^{2n+2} = lfp\, \lambda X.\, \dot{X}^{2n+1} \sqcap \mathcal{F}(X)$.

The Analyze procedure is defined as follows:

$$\textsc{Analyze}(P, i \mapsto o) = \{p \mapsto \mathcal{A}(p) \mid p \in Pos(P), \mathcal{A} = \lim_{n \to \infty} \dot{X}^n\}.$$

The procedure $\mathsf{Infeasible}(P, \mathcal{A}, i \mapsto o)$ on line 11 in Algo. 1 checks if any position $p \in Pos(P)$ is determined to be unreachable from the entry of the program P (i.e., $\mathsf{Infeasible}(P, \mathcal{A}, i \mapsto o) = \exists p \in Pos(P).\, \mathcal{A}(p) = \bot_{\hat{D}}$).

4.3 Meta Analysis Analyze$^{\#}$

In this section, we define abstract domains for meta analysis that can be used to analyze the behavior of analysis for pruning and derive the best hole-filling strategy.

Suppose that the abstract domain for the underlying analysis is $\hat{D}$. The meta abstract domain $\hat{D}^{\#} = \{\bot, \mathtt{Must}\top, \mathtt{May}\not\top\}$ where $\bot$ is the bottom element, $\mathtt{Must}\top$ represents the top element of the underlying abstract domain $\hat{D}$, and $\mathtt{May}\not\top$ represents elements that may not be the top element of $\hat{D}$ (also may be the top element hence $\mathtt{May}\not\top$ represents the set of all elements in $\hat{D}$). The galois connection $\mathcal{P}\left(\hat{D}\right) \underset{\alpha_{\hat{D}^{\#}}}{\overset{\gamma_{\hat{D}^{\#}}}{\leftrightarrows}} \hat{D}^{\#}$ between the meta abstract domain $\hat{D}^{\#}$ and the underlying abstract domain $\hat{D}$ is specified by the following concretization function: $\gamma_{\hat{D}^{\#}}(\bot) = \emptyset$, $\gamma_{\hat{D}^{\#}}(\mathtt{Must}\top) = \{\top\}$, and $\gamma_{\hat{D}^{\#}}(\mathtt{May}\not\top) = \hat{D}$.

The meta abstract interpreter $\textsc{Analyze}^{\#}(P, pos)$ also performs the iterative forward-backward analysis to obtain a map $\mathcal{A}^{\#}$ that maps each position in P to an abstract value in $\hat{D}^{\#}$. The forward abstract semantics is characterized by the least fixpoint of the following function $\mathcal{F}^{\#} : (Pos(P) \to \hat{D}^{\#}) \to (Pos(P) \to \hat{D}^{\#})$:

$$\mathcal{F}^{\#} = \lambda X.\, \mathcal{I}^{\#}_{\mathcal{F}} \sqcup F^{\#\#}(X) \text{ where } \mathcal{I}^{\#}_{\mathcal{F}} = \{p \mapsto \begin{cases} \mathtt{May}\not\top & (P\mid_p \text{ is a constant or } p = pos) \\ \bot & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}.$$

Each constant is mapped to $\mathtt{May}\not\top$ since it may not be the top element of the underlying abstract domain $\hat{D}$. Note that the position pos is also mapped to $\mathtt{May}\not\top$ since it is a hypothetical position to be filled, and thus it may not be the top element of the underlying abstract domain $\hat{D}$. Nonterminals (except pos) are mapped to $\mathtt{Must}\top$ by $F^{\#\#}$.

The forward abstract function $F^{\#\#}$ is defined as follows:

$$F^{\#\#}(X) = \{p \mapsto \begin{cases} \texttt{Must}\top & (P\,|_p \in N), p \neq pos \\ \overrightarrow{f}^{\#\#}(X(p\cdot 1), \cdots, X(p\cdot k)) & (P\,|_p = f(\cdots), arity(f) = k) \\ \bot & \text{(otherwise)} \end{cases} \mid p \in Pos(P)\}$$

where $\overrightarrow{f}^{\#\#}$ denotes the forward abstract operator of an operator f.

The backward abstract semantics is characterized by the least fixpoint of the following function $\mathcal{B}^{\#} : (Pos(P) \to \hat{D}^{\#}) \to (Pos(P) \to \hat{D}^{\#})$:

$$\mathcal{B}^{\#} = \lambda X.\, \mathcal{I}^{\#}_{\mathcal{B}} \sqcup B^{\#\#}(X) \quad \text{where} \quad \mathcal{I}^{\#}_{\mathcal{B}} = \{p \mapsto \begin{cases} \texttt{May}\not\top & (p = \epsilon) \\ \texttt{Must}\top & \text{(otherwise)} \end{cases} \mid p \in Pos(P)\}.$$

The final state $\mathcal{I}^{\#}_{\mathcal{B}}$ maps the root position to $\texttt{May}\not\top$ since the output of the root position should overapproximate the abstraction of the output example (i.e., $\alpha_{\hat{D}}(\{o\})$) and every other position to $\texttt{Must}\top$. The backward abstract function $B^{\#\#}$ is defined as follows:

$$B^{\#\#}(X) = \{p \mapsto \begin{cases} \overleftarrow{f}^{\#\#}_i(X(p'), X(p'\cdot 1), \cdots, X(p'\cdot k)) & (p = p'\cdot i, P\,|_{p'} = f(\cdots), arity(f) = k) \\ \texttt{Must}\top & \text{(otherwise)} \end{cases} \mid p \in Pos(P)\}$$

where $\overleftarrow{f}^{\#\#}_i$ denotes the backward abstract operator of an operator f. It takes an abstract value of the result and abstract values of arguments, and returns an abstract value of the i-th argument, which corresponds to the *necessary precondition* for the i-th argument to satisfy the input-output example.

As done in the underlying abstract interpreter, we compute the intersection of the forward and backward abstract semantics by obtaining the limit of the following increasing chain defined for all $n \in \mathbb{N}$ until the chain converges: $\dot{X}^{\#^0} = lfp\ \mathcal{F}^{\#}$, $\dot{X}^{\#^{2n+1}} = lfp\ \lambda X.\, \dot{X}^{\#^{2n}} \sqcup \mathcal{B}^{\#}(X)$, and $\dot{X}^{\#^{2n+2}} = lfp\ \lambda X.\, \dot{X}^{\#^{2n+1}} \sqcup \mathcal{F}^{\#}(X)$. The meta abstract interpreter $\text{Analyze}^{\#}$ is defined as follows:

$$\text{Analyze}^{\#}(P, i \mapsto o) = \left\{ p \mapsto \langle \mathcal{A}^{\#}_F(p), \mathcal{A}^{\#}_B(p) \rangle \;\middle|\; \begin{array}{l} p \in Pos(P), \mathcal{A}^{\#}_F = \mathcal{F}^{\#}(\mathcal{A}^{\#}), \\ \mathcal{A}^{\#}_B = \mathcal{B}^{\#}(\mathcal{A}^{\#}) \end{array} \right\}$$

where $\mathcal{A}^{\#} = \lim_{n\to\infty} \dot{X}^{\#^n}$.

The notable difference from the underlying abstract interpreter is that the meta abstract interpreter computes a pair of abstract values $\langle \mathcal{A}^{\#}_F(p), \mathcal{A}^{\#}_B(p) \rangle$ for each position $p \in Pos(P)$, where $\mathcal{A}^{\#}_F$ is an overapproximation of the results of the forward analysis and $\mathcal{A}^{\#}_B$ is an overapproximation of the results of the backward analysis computed during the iterative forward-backward analyses. The reason we compute a pair of abstract values is as follows: the underlying abstract interpreter may determine infeasibility of a program by checking if the analysis result is $\bot$ (i.e., unreachable) for some position $p \in Pos(P)$. Such the bottom element can be derived by the intersection of the forward and backward analyses (e.g., assuming that our underlying abstract domain is the interval domain, if the forward analysis computes $[1, 2]$ and the backward analysis computes $[3, 4]$ for a position, then the intersection is $\bot$ since there is no common value between the two sets). However, the meta abstract interpreter does not compute the bottom element $\bot$ for any position because it is based on the coarse abstract domain that does not keep track of any information about the concrete values of the program. Therefore, we approximate the bottom element in the underlying analysis by the pair $\langle \texttt{May}\not\top, \texttt{May}\not\top \rangle$ in the meta analysis, which indicates that the position may not be the top element of the underlying abstract domain $\hat{D}$ in both forward and backward analyses (i.e., $\bot_{\hat{D}} \in \gamma_{\hat{D}^{\#}}(\texttt{May}\not\top \sqcap \texttt{May}\not\top)$).

Example 4.1. We illustrate how the meta abstract interpreter $\text{Analyze}^{\#}$ works for the overview example in §2. Suppose we have the following forward abstract operator for string concatenation

$\texttt{ConCat}$: $\overrightarrow{\texttt{ConCat}}^{\#\#}(a_1^\#, a_2^\#, a_3^\#) = a_3^\#$ where $a_1^\#, a_2^\#, a_3^\# \in \hat{D}^\#$ are abstract values for the first, second, and third arguments of $\texttt{ConCat}$, respectively. The operator simply returns the abstract value of the third argument as the result. That is because if the third argument is the top (non-top resp.) element in the underlying analysis, then the result is also the top (non-top resp.) element regardless of the values of the first and second arguments. Similarly, suppose we have the following backward abstract operator for $\texttt{ConCat}$:

$$\overleftarrow{\texttt{ConCat}}_1^{\#\#}(a_2^\#, a_3^\#, a_r^\#) = \begin{cases} \texttt{May}\not\top & \text{if } a_2^\# \neq \texttt{Must}\top \wedge a_r^\# \neq \texttt{Must}\top \\ \texttt{Must}\top & \text{otherwise} \end{cases} \qquad \begin{array}{l} \overleftarrow{\texttt{ConCat}}_2^{\#\#}(a_1^\#, a_3^\#, a_r^\#) = \texttt{Must}\top \\ \overleftarrow{\texttt{ConCat}}_3^{\#\#}(a_1^\#, a_2^\#, a_r^\#) = \texttt{Must}\top \end{array}$$

Next, we describe how the meta abstract interpreter $\textsc{Analyze}^\#$ analyzes the candidate programs (4), (5), and (6) with hypothetical hole-fillings in §2.

For the candidate program (4) with the hypothetical filling of the hole at position 1, the initial and final states are as follows:

$$\mathcal{I}_{\mathcal{F}}^\# = \{\epsilon \mapsto \bot, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \bot, 3 \mapsto \bot\} \qquad \mathcal{I}_{\mathcal{B}}^\# = \{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}.$$

With these initial and final states, $\dot{X}^{\#^0} = \mathit{lfp}\ \mathcal{F}^\#$ is computed as $\{\epsilon \mapsto \texttt{Must}\top, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}$. Next, $\dot{X}^{\#^1} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^0} \sqcup \mathcal{B}^\#(X)$ is computed as $\{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}$. Next, $\dot{X}^{\#^2} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^1} \sqcup \mathcal{F}^\#(X)$ is computed as the same as $\dot{X}^{\#^1}$, and thus the chain converges. Therefore, the meta abstract interpreter $\textsc{Analyze}^\#$ returns $\langle \mathcal{A}_F^\#, \mathcal{A}_B^\# \rangle$ where

$$\begin{aligned} \mathcal{A}_F^\# &= \mathcal{I}_{\mathcal{F}}^\# \sqcup \overrightarrow{F}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{Must}\top, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\} \\ \mathcal{A}_B^\# &= \mathcal{I}_{\mathcal{B}}^\# \sqcup \overleftarrow{B}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}. \end{aligned}$$

For the candidate program (5) with the hypothetical filling of the hole at position 2, the initial state $\mathcal{I}_{\mathcal{F}}^\#$ is $\{\epsilon \mapsto \bot, 1 \mapsto \bot, 2 \mapsto \texttt{May}\not\top, 3 \mapsto \bot\}$ and the final state is the same as the previous case. With these initial and final states, $\dot{X}^{\#^0} = \mathit{lfp}\ \mathcal{F}^\#$ is computed as $\{\epsilon \mapsto \texttt{Must}\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{May}\not\top, 3 \mapsto \texttt{Must}\top\}$. Next, $\dot{X}^{\#^1} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^0} \sqcup \mathcal{B}^\#(X)$ is computed as $\{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \texttt{May}\not\top, 3 \mapsto \texttt{Must}\top\}$. Next, $\dot{X}^{\#^2} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^1} \sqcup \mathcal{F}^\#(X)$ is computed as the same as $\dot{X}^{\#^1}$, and thus the chain converges. Therefore, the meta abstract interpreter $\textsc{Analyze}^\#$ returns $\langle \mathcal{A}_F^\#, \mathcal{A}_B^\# \rangle$ where

$$\begin{aligned} \mathcal{A}_F^\# &= \mathcal{I}_{\mathcal{F}}^\# \sqcup \overrightarrow{F}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{Must}\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{May}\not\top, 3 \mapsto \texttt{Must}\top\} \\ \mathcal{A}_B^\# &= \mathcal{I}_{\mathcal{B}}^\# \sqcup \overleftarrow{B}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{May}\not\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}. \end{aligned}$$

Lastly, for the candidate program (6) with the hypothetical filling of the hole at position 3, the initial state $\mathcal{I}_{\mathcal{F}}^\#$ is $\{\epsilon \mapsto \bot, 1 \mapsto \bot, 2 \mapsto \bot, 3 \mapsto \texttt{May}\not\top\}$ and the final state is the same as the previous case. With these initial and final states, $\dot{X}^{\#^0} = \mathit{lfp}\ \mathcal{F}^\#$ is computed as $\{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{May}\not\top\}$. Next, $\dot{X}^{\#^1} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^0} \sqcup \mathcal{B}^\#(X)$ is computed as $\{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{May}\not\top\}$. Next, $\dot{X}^{\#^2} = \mathit{lfp}\ \lambda X.\ \dot{X}^{\#^1} \sqcup \mathcal{F}^\#(X)$ is computed as the same as $\dot{X}^{\#^1}$, and thus the chain converges. Therefore, the meta abstract interpreter $\textsc{Analyze}^\#$ returns $\langle \mathcal{A}_F^\#, \mathcal{A}_B^\# \rangle$ where

$$\begin{aligned} \mathcal{A}_F^\# &= \mathcal{I}_{\mathcal{F}}^\# \sqcup \overrightarrow{F}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{May}\not\top\} \\ \mathcal{A}_B^\# &= \mathcal{I}_{\mathcal{B}}^\# \sqcup \overleftarrow{B}^{\#\#}(\dot{X}^{\#^2}) = \{\epsilon \mapsto \texttt{May}\not\top, 1 \mapsto \texttt{Must}\top, 2 \mapsto \texttt{Must}\top, 3 \mapsto \texttt{Must}\top\}. \end{aligned}$$

The soundness of the meta abstract interpreter is guaranteed by the following theorem.

Theorem 4.2. *For a given example-based SyGuS problem $\langle G, \mathcal{E} \rangle$, for every program $P \in L(G)$, position $p \in Pos(P)$, and hole position $pos \in Holes(P)$, if $F^{\#\#}$ and $B^{\#\#}$ are sound with respect to $F^\#$ and $B^\#$ respectively, then $\textsc{Analyze}(P, \mathcal{E})(p) \in \gamma_{\hat{D}^\#}(\mathcal{A}^\#)$ where $\mathcal{A}^\# = \lim_{n\to\infty} \dot{X}^{\#^n}$.*

Proof. Available in the supplementary material. □

4.4 Determination of Rank of Positions

The meta abstract interpreter Analyze$^\#$ computes a pair of abstract values $\mathcal{A}^{\text{Meta}}(p) = \langle \mathcal{A}^\#_F(p), \mathcal{A}^\#_B(p)\rangle$ for each position $p \in \mathit{Pos}(P)$ when $\mathit{pos} \in \mathsf{Holes}(P)$ is hypothetically filled. Using the pair of abstract values, we can determine the rank of a position as follows:

- **Rank-1 position**: A hole position *pos* is a Rank-1 position if filling the hole position may lead to an infeasible candidate, which is determined by the condition
$$\exists p \in \mathit{Pos}(P).\ \mathcal{A}^{\text{Meta}}(p) = \langle \mathsf{May}\not\top, \mathsf{May}\not\top\rangle. \tag{7}$$
This means that there exists a position having $\bot_{\hat{D}}$ in the underlying analysis. The bottom element can be derived only by the intersection of the forward and backward analyses neither of which computes the top element at the position.
- **Rank-2 position**: A hole position *pos* is a Rank-2 position if filling the hole position may prune components, which is determined by the condition
$$\exists p \in \mathsf{Holes}(P) \backslash \{\mathit{pos}\}.\ \mathcal{A}^{\text{Meta}}(p) = \langle \mathtt{Must}\top, \mathsf{May}\not\top\rangle. \tag{8}$$
This means that there exists a hole position p' other than *pos* (which is already filled hypothetically) for which the underlying backward analysis computes a non-top element, indicating that filling the position with any component may lead to a candidate that has non-trivial necessary preconditions over the other holes.

The procedures $\mathsf{MayInfeasible}(\mathcal{A}^{\text{Meta}})$ and $\mathsf{MayPruneComponents}(\mathcal{A}^{\text{Meta}})$ in Algo. 2 check the conditions (7) and (8), respectively.

Example 4.3. In Example 4.1, for the candidate program (6) with the hypothetical filling of the hole at position 3, $\mathcal{A}^{\text{Meta}}(\epsilon) = \langle \mathsf{May}\not\top, \mathsf{May}\not\top\rangle$, thereby satisfying the condition (7). Therefore, the position 3 is determined to be a Rank-1 position. For the candidate program (5) with the hypothetical filling of the hole at position 2, $\mathcal{A}^{\text{Meta}}(1) = \langle \mathtt{Must}\top, \mathsf{May}\not\top\rangle$, thereby satisfying the condition (8). Therefore, the position 2 is determined to be a Rank-2 position. For the candidate program (4) with the hypothetical filling of the hole at position 1, neither condition (7) nor (8) is satisfied. Therefore, the position 1 is not assigned any rank.

If the meta abstract interpreter Analyze$^\#$ is sound with respect to the underlying abstract interpreter Analyze, then the following theorem holds:

Theorem 4.4. *If the meta abstract interpretation $\mathcal{A}^\#$ is sound with respect to the underlying abstract interpretation $\mathcal{A}$, then the procedure PickHole is guaranteed to be an admissible hole-filling strategy.*

Proof. Available in the supplementary material. □

4.5 Optimizations

4.5.1 Filling Multiple Holes. So far, for ease of exposition, we have described the hole-filling strategy for dealing with a single hole at a time. However, in practice, it is often beneficial to consider multiple holes at once to improve the efficiency of the analysis.

To see the necessity of filling multiple holes simultaneously, consider a partial program of form $\square + \square + \square$ where $\square$ can be filled with any integer expression. No matter which hole is filled first, the underlying analysis will not be able to neither prune components for the other remaining holes nor determine infeasibility of candidates; candidates with a single hole filled have too many possibilities in their behavior. If we fill two holes simultaneously, however, we can infer a necessary precondition for the third hole, which can lead to pruning components for the hole.

To choose multiple holes to fill simultaneously, we modify Algo. 2 to consider subsets of holes rather than individual holes at each iteration of the loop in lines 4–11. Instead of iterating through single holes $h \in \mathsf{Holes}(P)$, the loop iterates through subsets $H \subseteq \mathsf{Holes}(P)$ of increasing size $|H| = 1, 2, \ldots$ until the maximum size of the subset is reached. The meta analyzer $\textsc{Analyze}^{\#}$ is then called with the program P and the subset H of holes to analyze the behavior of the program with hypothetical filling of all holes in H. In addition, Algo. 2 keeps track of a set of sets of holes instead of a set of holes and returns a set of holes as output. In our implementation, we use the maximum number of holes to be filled simultaneously as a parameter of the algorithm and we set it to 3 by default, which is sufficient for most cases.

4.5.2 Incremental Meta Analysis. Instead of starting from scratch, our meta analyzer reuses previously computed intermediate results and selectively updates only those parts of the program that are affected by the changes. Because the changes are hypothetical fillings of holes, the meta analyzer updates the analysis results only for the holes that are hypothetically filled in the current iteration and computes the analysis results starting from the holes using the worklist algorithm.

4.6 Instantiations to String and Bitvector Manipulation

In this section, we present how to instantiate the generic hole-filling strategy to the domains of bitvectors and strings, which are commonly used in program synthesis tasks.

Because of the space constraints, we only present some noteworthy points of our abstract domains. The full details of our abstract domains are available in the supplementary material.

4.6.1 Notations. For the theory of strings and bitvectors, we follow the standard syntax and semantics of the string operators described in the SMT-LIB v2.0 standard [3]. For an interval $[l, h]$, we denote $\mathsf{lb}([l, h])$ and $\mathsf{ub}([l, h])$ as the lower and upper bounds of $[l, h]$ respectively. $\overline{[l, h]}$ denotes $h - l$. Also, we will denote $\oplus_{\mathbb{I}}$ as the standard interval operation for an operation $\oplus$ on integers.

4.6.2 Abstract Domain for Fixed-Width Bitvectors.

Abstract Domain for the Underlying Abstract Interpretation. We use the same abstract domain as in the prior work [38]. The domain is a reduced product of three abstract domains: bitwise domain for tracking the value of each bit of a bit-vector independently, and signed / unsigned interval domains for representing intervals of signed / unsigned bit-vectors. We refer the reader to the prior work [38] for more details.

Abstract Domain for Meta Analysis. The meta abstract domain for the bitvector domain is $\hat{D}^{\#}$ described in §4.3.

The forward and backward abstract operators for the meta abstract domain are described in Fig. 1 and Fig. 2, respectively. In the following, we explain some salient features of the operators.

- The forward abstract operators for `bvand` and `bvor` return $\mathtt{May}\not\top$ if either of the inputs is $\mathtt{May}\not\top$, and $\mathtt{Must}\top$ otherwise. This is because the forward abstract operators in the underlying analysis for these functions may yield a non-top abstract value even if either of the inputs is top, and the output is top only if both inputs are top. This is captured by the meta abstract operators as well.
- The backward abstract operator for `bvand` in the bitwise domain captures the behavior of the underlying analysis as follows: For example, for each abstract bit of the result, if the bit is 1, we can infer that the corresponding bit of the first operand is 1 as well. If the result is non-top, we can infer that the first operand is non-top as well.

The following theorem states that the meta abstract operators are sound with respect to the underlying abstract operators in the bitvector domain.

For $f \in \{\mathsf{bvneg}, \mathsf{bvnot}\}$

$$\overrightarrow{f}^{\#\#}_{\hat{D}^\#}(a^\#) = \begin{cases} \mathsf{May}\not\top & (a^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

For $f \in \{\mathsf{bvxor}, \mathsf{bvadd}, \mathsf{bvsub}\}$

$$\overrightarrow{f}^{\#\#}_{\hat{D}^\#}(a^\#, b^\#) = \begin{cases} \mathsf{May}\not\top & (a^\# = \mathsf{May}\not\top \land b^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

For $f \in \{\mathsf{bvand}, \mathsf{bvor}, \mathsf{bvmul}, \mathsf{bvsdiv}, \mathsf{bvlshr}, \mathsf{bvshl}\}$

$$\overrightarrow{f}^{\#\#}_{\hat{D}^\#}(a^\#, b^\#) = \begin{cases} \mathsf{May}\not\top & (a^\# = \mathsf{May}\not\top \lor b^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

For $f \in \{\mathsf{bvurem}, \mathsf{bvsrem}\}$

$$\overrightarrow{f}^{\#\#}_{\hat{D}^\#}(a^\#, b^\#) = \begin{cases} \mathsf{May}\not\top & (b^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

Fig. 1. Forward abstract operators in the bitvector domain for meta analysis

For $f \in \{\mathsf{bvand}, \mathsf{bvor}\}$ ($\overleftarrow{f}^{\#\#}_{\hat{D}^\#,2}$ is similar.)

$$\overleftarrow{f}^{\#\#}_{\hat{D}^\#,1}(r^\#, b^\#) = \begin{cases} \mathsf{May}\not\top & (r^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

For $f \in \{\mathsf{bvxor}, \mathsf{bvadd}, \mathsf{bvsub}\}$ ($\overleftarrow{f}^{\#\#}_{\hat{D}^\#,2}$ is similar.)

$$\overleftarrow{f}^{\#\#}_{\hat{D}^\#,1}(r^\#, b^\#) = \begin{cases} \mathsf{May}\not\top & (r^\# = \mathsf{May}\not\top \land b^\# = \mathsf{May}\not\top) \\ \mathsf{Must}\top & (\text{otherwise}) \end{cases}$$

Fig. 2. Backward abstract operators in the bitvector domain for meta analysis (selected)

Theorem 4.5. *The meta abstract operators in Fig. 1 and Fig. 2 are sound with respect to the underlying abstract operators in the bitvector domain.*

Proof. Available in the supplementary material. □

4.6.3 Abstract Domains for Strings.

Abstract Domain for Underlying Analysis. Our abstract domain for strings is a also reduced product domain. Let Σ be a finite set of characters, and $\hat{\Sigma} = \Sigma \cup \{*\}$ be the set of characters including a wildcard character $*$ that represents an unknown character. The following set will be used to represent the abstract domain for strings:

$$\hat{C} \stackrel{\text{def}}{=} \{\langle s, i\rangle \mid s \in \hat{\Sigma}^*, i \in \mathbb{I}\}$$

where $\mathbb{I}$ is the set of intervals of the form $[l, h]$ where $l \in \mathbb{N}$ and $h \in \mathbb{N} \cup \{\infty\}$. The first component s represents a prefix or suffix of strings, and the second component i represents an upperbound on the length of the strings, hence the lowerbound cannot be a negative number.

Our abstract domain is a reduced product of the following domains.

- *Prefix domain with length* ($\langle \hat{P}, \sqsubseteq_{\hat{P}}, \sqcup_{\hat{P}}, \sqcap_{\hat{P}}\rangle$): a domain that tracks the prefix of a string and an upperbound on the length of the string where $\hat{P} = \hat{C}$. Each abstract element is of the form (s, i) where $s \in \Sigma^*$ represents the prefix of strings it abstracts and $i \in \mathbb{I}$ represents the range of the length of the strings it abstracts. The galois connection $\mathcal{P}(\Sigma^*) \underset{\alpha_{\hat{P}}}{\overset{\gamma_{\hat{P}}}{\longleftrightarrow}} \hat{P}$ is defined as follows:
$\alpha_{\hat{P}}(S) \stackrel{\text{def}}{=} (\mathsf{LCP}(S), [\ell_{\min}, \ell_{\max}])$, $\gamma_{\hat{P}}(s, i) \stackrel{\text{def}}{=} \begin{cases} \emptyset & (i = [0, 0]) \\ \{s[\epsilon/*] \cdot t \mid t \in \hat{\Sigma}^*\} & (\mathsf{ub}(i) = \infty) \\ \{s[\epsilon/*] \cdot t \mid \mathsf{lb}(i) \le |t| \le \mathsf{ub}(i), t \in \Sigma^*\} & (\text{otherwise}) \end{cases}$
where $\mathsf{LCP}(S)$ is the longest common prefix of the strings in S and $\ell_{\min} = \min_{s \in S} |s|$ and $\ell_{\max} = \max_{s \in S} |s|$.
- *Suffix domain with length* ($\langle \hat{S}, \sqsubseteq_{\hat{S}}, \sqcup_{\hat{S}}, \sqcap_{\hat{S}}\rangle$): a domain that tracks the suffix of a string and an upperbound on the length of the string where $\hat{S} = \hat{C}$. The domain is similar to the prefix domain with length with the only difference that it tracks the suffix of a string instead of the prefix. The galois connection $\mathcal{P}(\Sigma^*) \underset{\alpha_{\hat{S}}}{\overset{\gamma_{\hat{S}}}{\longleftrightarrow}} \hat{S}$ is defined as follows: $\alpha_{\hat{S}}(S) \stackrel{\text{def}}{=}$ $(\mathsf{LCS}(S), [\ell_{\min}, \ell_{\max}])$, $\gamma_{\hat{S}}(s, i) \stackrel{\text{def}}{=} \begin{cases} \emptyset & (i = [0, 0]) \\ \{t \cdot s[\epsilon/*] \mid t \in \Sigma^*\} & (\mathsf{ub}(i) = \infty) \\ \{t \cdot s[\epsilon/*] \mid \mathsf{lb}(i) \le |t| \le \mathsf{ub}(i), t \in \Sigma^*\} & (\text{otherwise}) \end{cases}$
where $\mathsf{LCS}(S)$ is the longest common suffix of the strings in S.

$$\begin{aligned}
\overrightarrow{\texttt{ConCat}}^{\#}_{\hat{P}}(\langle p_1, i_1\rangle, \langle p_2, i_2\rangle) &= \begin{cases} (p_1 \cdot p_2, i_1 +_{\mathbb{I}} i_2) & (\text{if } i_1 \text{ is constant}) \\ (p_1, i_1 +_{\mathbb{I}} i_2) & (\text{otherwise}) \end{cases} \\
\overrightarrow{\texttt{CharAt}}^{\#}_{\hat{P}}(\langle p_1, i_1\rangle, [l,u]) &= \begin{cases} \langle \alpha_{\hat{P}}(\{\texttt{CharAt}(p_1,k) \mid l \le k \le u\}), [1,1]\rangle & (\text{if } u < \mathsf{lb}(i_1)) \\ \langle \epsilon, [1,1]\rangle & (\text{otherwise}) \end{cases} \\
\overrightarrow{\texttt{StrToInt}}^{\#}_{\hat{P}}(\langle p, i\rangle) &= [l,u] \\
&\quad \text{where } l = \texttt{StrToInt}(p[\epsilon/*] \cdot \texttt{StrToInt}(0)^{\mathsf{lb}(i)-|s|}), \\
&\qquad u = (\texttt{StrToInt}(p[\epsilon/*]) + 1) \times 10^{\mathsf{ub}(i)-|s|} - 1
\end{aligned}$$

Fig. 3. Forward abstract operators in the prefix domain (selected)

$$\begin{aligned}
\overleftarrow{\texttt{ConCat}}^{\#}_{\hat{P},1}(\langle p, i\rangle, \langle p_2, i_2\rangle) &= \langle \texttt{SubStr}(p, 0, \mathsf{lb}(i) - \mathsf{ub}(i_2)), i -_{\mathbb{I}} i_2\rangle \\
\overleftarrow{\texttt{ConCat}}^{\#}_{\hat{P},2}(\langle p, i\rangle, \langle p_1, i_1\rangle) &= \begin{cases} \langle \texttt{SubStr}(p, 0, \mathsf{lb}(i) - \mathsf{ub}(i_1)), i -_{\mathbb{I}} i_1\rangle & (\text{if } i_1 \text{ is constant}) \\ \langle \epsilon, i -_{\mathbb{I}} i_1\rangle & (\text{otherwise}) \end{cases} \\
\overleftarrow{\texttt{CharAt}}^{\#}_{\hat{P},1}(\langle p, i\rangle, [l,u]) &= \begin{cases} \langle *^{l} \cdot p, [l + \mathsf{lb}(i), \infty]\rangle & (\text{if } l = u) \\ \langle \epsilon, [l, \infty]\rangle & (\text{otherwise}) \end{cases} \\
\overleftarrow{\texttt{CharAt}}^{\#}_{\hat{P},2}(\langle p, i\rangle, \langle p_1, i_1\rangle) &= \begin{cases} [\min I, \max I] & (\text{if } i_1 \text{ is constant}) \\ [\texttt{IndexOf}^{*}(p_1, p, 0), \mathsf{ub}(i) - 1] & (\text{otherwise}) \end{cases} \\
&\quad \text{where } I = \{\texttt{IndexOf}^{*}(p_1, p, k) \mid 0 \le k < |p_1|\} \\
\overleftarrow{\texttt{StrToInt}}^{\#}_{\hat{P},1}([l,u]) &= \begin{cases} \langle p, [|p|, |u_s|]\rangle & (|l_s| = |u_s|) \\ \langle \epsilon, [|l_s|, |u_s|]\rangle & (\text{otherwise}) \end{cases} \\
&\quad \text{where } l_s = \texttt{IntToStr}(l), u_s = \texttt{IntToStr}(u), \text{ and } p = \mathsf{LCP}(l_s, u_s)
\end{aligned}$$

Fig. 4. Backward abstract operators in the prefix domain (selected)

The above two domains are combined to form the product abstract domain $\hat{D} = \hat{P} \times \hat{S}$. The galois connection $\mathcal{P}(\Sigma^*) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{D}$ can be constructed in a straightforward manner by combining the galois connections of the individual domains.

Now we define forward and backward abstract operators. Any cases not mentioned in the full definitions are handled by the default operator which returns the bottom element of the domain if any of the arguments is the bottom element and returns the top element otherwise.

Fig. 3 and Fig. 4 depict the forward and backward abstract operators for the prefix domain, respectively. Some features of the backward abstract operators are as follows:

- The forward abstract operator $\overrightarrow{\texttt{ConCat}}^{\#}_{\hat{P}}$ concatenates two prefixes and add their intervals. If the first prefix is constant, it adds the interval of the second prefix to the interval of the first prefix.
- The backward abstract operator $\overleftarrow{\texttt{StrToInt}}^{\#}_{\hat{P}}$ takes a given interval result $[l,u]$ to a prefix p such that the prefix p is the longest common prefix of the string representations of the lower and upper bounds of the interval l and u.

Abstract Domain for the Meta Analysis. We will denote $D^{\#}$ as the abstract domain for our meta analysis for a domain D for underlying analysis.

The domain $\hat{D}^{\#}$ for our meta analysis for strings is defined as a reduced product $\hat{P}^{\#} \times \hat{S}^{\#}$ where both $\hat{P}^{\#}$ and $\hat{S}^{\#}$ are defined as $\hat{C}^{\#} \times \mathbb{I}^{\#}$ where $\hat{C}^{\#} \overset{\text{def}}{=} \{\bot, \mathsf{Must}\top, \mathsf{May}\not\top\}$ and $\mathbb{I}^{\#} \overset{\text{def}}{=} \{\bot, \mathsf{Con}, \mathsf{Bnd}, \mathsf{Unb}\}$. The domain $\hat{C}^{\#}$ is the set of abstract elements for the prefix and suffix domains, and $\mathbb{I}^{\#}$ is the set of abstract elements for the length domain. The order between the abstract elements is defined as follows: $\bot \sqsubseteq_{\hat{C}^{\#}} \mathsf{Must}\top \sqsubseteq_{\hat{C}^{\#}} \mathsf{May}\not\top$ and $\bot \sqsubseteq_{\mathbb{I}^{\#}} \mathsf{Con} \sqsubseteq_{\mathbb{I}^{\#}} \mathsf{Bnd} \sqsubseteq_{\mathbb{I}^{\#}} \mathsf{Unb}$. The galois connections $\mathcal{P}\left(\hat{P}\right) \underset{\alpha_{\hat{P}^{\#}}}{\overset{\gamma_{\hat{P}^{\#}}}{\leftrightarrows}} \hat{P}^{\#}$ and $\mathcal{P}\left(\hat{S}\right) \underset{\alpha_{\hat{S}^{\#}}}{\overset{\gamma_{\hat{S}^{\#}}}{\leftrightarrows}} \hat{S}^{\#}$ is defined as follows (only the concretization functions

$$\begin{aligned}
\overrightarrow{\texttt{ConCat}}^{\#\#}_{\hat{P}^\#}(\langle p_1^\#, i_1^\#\rangle, \langle p_2^\#, i_2^\#\rangle) &= \begin{cases} (p_1^\# \odot p_2^\#, i_1^\# +_{\mathbb{I}^\#} i_2^\#) & (i_1^\# = \texttt{Con}) \\ (p_1^\#, i_1^\# +_{\mathbb{I}^\#} i_2^\#) & (\text{otherwise}) \end{cases} \\
\overrightarrow{\texttt{CharAt}}^{\#\#}_{\hat{P}^\#}(\langle p_1^\#, i_1^\#\rangle, i^\#) &= \begin{cases} \langle \texttt{May}\not\top, \texttt{Con}\rangle & (p_1^\# = \texttt{May}\not\top \wedge i^\# = \texttt{Con}) \\ \langle \texttt{Must}\top, \texttt{Con}\rangle & (\text{otherwise}) \end{cases} \\
\overrightarrow{\texttt{StrToInt}}^{\#\#}_{\hat{P}^\#}(\langle p^\#, i^\#\rangle) &= i^\# \sqcup_{\mathbb{I}^\#} \texttt{Bnd}
\end{aligned}$$

Fig. 5. Forward abstract operators in the prefix domain for meta analysis (selected)

$$\begin{aligned}
\overleftarrow{\texttt{ConCat}}^{\#\#}_{\hat{P}^\#,1}(\langle p^\#, i^\#\rangle, \langle p_2^\#, i_2^\#\rangle) &= \begin{cases} \langle \texttt{Must}\top, i^\# -_{\mathbb{I}^\#} i_2^\#\rangle & (i_2^\# = \texttt{Unb}) \\ \langle p^\#, i^\# -_{\mathbb{I}^\#} i_2^\#\rangle & (\text{otherwise}) \end{cases} \\
\overleftarrow{\texttt{ConCat}}^{\#\#}_{\hat{P}^\#,2}(\langle p^\#, i^\#\rangle, \langle p_1^\#, i_1^\#\rangle) &= \begin{cases} \langle \texttt{Must}\top, i^\# -_{\mathbb{I}^\#} i_1^\#\rangle & (i_1^\# = \texttt{Unb}) \\ \langle p^\#, i^\# -_{\mathbb{I}^\#} i_1^\#\rangle & (\text{otherwise}) \end{cases} \\
\overleftarrow{\texttt{CharAt}}^{\#\#}_{\hat{P}^\#,1}(\langle p^\#, i^\#\rangle, i_1^\#) &= \begin{cases} \langle \texttt{May}\not\top, \texttt{Unb}\rangle & (p^\# = \texttt{May}\not\top \wedge i_1^\# = \texttt{Con}) \\ \langle \texttt{Must}\top, \texttt{Unb}\rangle & (\text{otherwise}) \end{cases} \\
\overleftarrow{\texttt{CharAt}}^{\#\#}_{\hat{P}^\#,2}(\langle p^\#, i^\#\rangle, \langle p_1^\#, i_1^\#\rangle) &= i^\# \sqcup_{\mathbb{I}^\#} \texttt{Bnd} \\
\overleftarrow{\texttt{StrToInt}}^{\#\#}_{\hat{P}^\#,1}(\langle p^\#, i^\#\rangle) &= \begin{cases} \langle \texttt{May}\not\top, \texttt{Con}\rangle & (i^\# = \texttt{Con}) \\ \langle \texttt{May}\not\top, \texttt{Bnd}\rangle & (i^\# = \texttt{Bnd}) \\ \langle \texttt{Must}\top, \texttt{Unb}\rangle & (\text{otherwise}) \end{cases}
\end{aligned}$$

Fig. 6. Backward abstract operators in the prefix domain for meta analysis (selected)

are shown, assuming $\hat{P}/\hat{S}$ consistently denotes either $\hat{P}$ or $\hat{S}$):

$$\gamma_{\hat{P}^\#/\hat{S}^\#}(\bot) \stackrel{\text{def}}{=} \{\bot\}, \quad \gamma_{\hat{P}^\#/\hat{S}^\#}(\texttt{Must}\top) \stackrel{\text{def}}{=} \{\langle \epsilon, [0,\infty]\rangle\}, \quad \gamma_{\hat{P}^\#/\hat{S}^\#}(\texttt{May}\not\top) \stackrel{\text{def}}{=} \hat{P}/\hat{S}$$
$$\gamma_{\mathbb{I}^\#}(\bot) \stackrel{\text{def}}{=} \{\bot\}, \quad \gamma_{\mathbb{I}^\#}(\texttt{Con}) \stackrel{\text{def}}{=} \{[n,n] \mid n \in \mathbb{N}\},$$
$$\gamma_{\mathbb{I}^\#}(\texttt{Bnd}) \stackrel{\text{def}}{=} \{[l,u] \mid l \le u, l,u \in \mathbb{N}\}, \quad \gamma_{\mathbb{I}^\#}(\texttt{Unb}) \stackrel{\text{def}}{=} \mathbb{I}$$

The above two domains are combined to form the product abstract domain $\hat{D}^\# = \hat{P}^\# \times \hat{S}^\#$. The galois connection $\mathcal{P}\left(\hat{D}\right) \underset{\alpha_{\hat{D}^\#}}{\overset{\gamma_{\hat{D}^\#}}{\leftrightarrows}} \hat{D}^\#$ can be constructed in a straightforward manner by combining the galois connections of the individual domains.

Fig. 5 and Fig. 6 depict the forward and backward abstract operators for the prefix domain in the meta analysis, respectively.

- The forward abstract operators $\overrightarrow{\texttt{ConCat}}^{\#\#}_{\hat{P}^\#}$ and $\overrightarrow{\texttt{ConCat}}^{\#\#}_{\hat{S}^\#}$ concatenate two prefixes/suffixes and add their intervals. If the length of the first prefix/suffix is constant, the two prefixes/-suffixes are concatenated and their length intervals are added (the operator $\odot$ simulates the concatenation of two prefixes/suffixes in the meta analysis, where $\odot$ is defined as follows: $a \odot b = \texttt{May}\not\top$ if $a = \texttt{May}\not\top$ or $b = \texttt{May}\not\top$, otherwise $a \odot b = \texttt{Must}\top$).
- The forward abstract operator $\overrightarrow{\texttt{StrToInt}}^{\#\#}_{\hat{P}^\#}$ reflect the fact that multiple string representations of integers of a constant or bounded length can be represented by a bounded interval. Therefore, if the length of the prefix is constant or a bounded interval, the result is a bounded interval. Otherwise, the result is an unbounded interval.

5 Evaluation

We have implemented our approach in a tool called Hope[7] which consists of 14K lines of OCaml code and employs Z3 [11] as the constraint solving engine.

This section evaluates Hope to answer the following questions:

Q1: How does Hope perform on synthesis tasks from a variety of different application domains?
Q2: How does Hope compare with existing state-of-the-art synthesis tools in terms of scalability?
Q3: How effective is the meta analysis in guiding the hole-filling order?

[7]**H**ole-filling **O**rder **P**rediction for **E**fficient synthesis

All of our experiments were run on a machine with an Intel i9-13900hx 2.2GHz CPU and 64GB of RAM within a 60-second timeout per synthesis problem.

5.1 Experimental Setup

Benchmarks. We collect benchmarks from two domains: string and bitvector manipulation. They are from the benchmarks used for evaluating the Duet, Simba, and SynthPhonia, which are the state-of-the-art synthesis tools for bitvector and string synthesis, respectively.

The string benchmarks comprise **337** problems that come from the existing two benchmark suites: Duet and Hardbench.

- The Duet benchmark contains 205 string manipulation problems, consisting of 108 problems from the SyGuS competition and 97 additional benchmarks from StackOverflow and ExcelJet.
- The Hardbench benchmark [13] includes 132 challenging string transformation problems involving heavy case-splitting.

There is another benchmark suite called Prose benchmark that has been used to evaluate FlashFill++ [6] and SynthPhonia [13], which contains 354 problems. However, we do not include it in our evaluation for the following reason. The problems require grammar constructs (beyond SyGuS interchange format (SyGuS-IF)) that our approach does not currently support such as date, time, and floating-point operations. Even if we restrict to problems using only supported constructs, the remaining problems are too easy for all solvers, including ours, to solve.

The bitvector benchmarks comprise **1,294** problems that come from three established benchmark suites: Bitvec-cond, Deobfusc, and Hacker's Delight.

- The Bitvec-cond benchmark contains 750 tasks of synthesizing conditional bitvector programs from input-output examples. The problems involve complex case-splitting.
- The Deobfusc benchmark [38] consists of 500 problems focused on deobfuscation tasks, where the goal is to synthesize simplified equivalent expressions for obfuscated bitvector operations.
- The Hacker's Delight (HD) benchmark includes 44 problems of synthesizing bit-twiddling hacks.

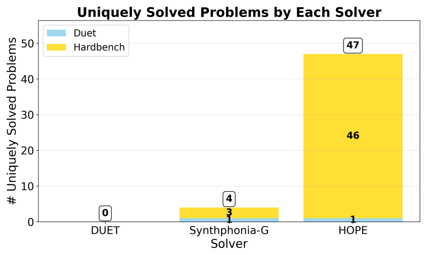
Compared Synthesizers. We compare Hope with state-of-the-art synthesis tools in both domains. For string synthesis, we compare Hope against: Duet and SynthPhonia-G.

- Duet employs a bidirectional search strategy that combines bottom-up enumeration with top-down propagation to recursively decompose a given synthesis problem into multiple subproblems.
- SynthPhonia-G is a grammar-constrained version of the full SynthPhonia tool, which implements a concurrent synthesis approach that orchestrates deductive and enumerative processes asynchronously. It supports not only various string operations but also date, time, and floating point operations. We limit the grammar of SynthPhonia-G to use only string operations compatible with SyGuS-IF. SynthPhonia can use multithreading, but we disable it to ensure a fair comparison with the other SyGuS-based tools.

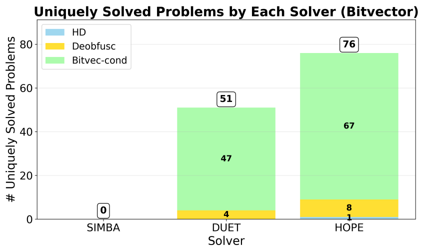
For bitvector synthesis, we compare Hope against two state-of-the-art tools: Duet and Simba.

- Simba employs a forward-backward analysis approach for bitvector synthesis. Because Hope's bitvector synthesis implementation is based on Simba, it can be regarded as a variant of Hope that does not use meta analysis.

While there is another state-of-the-art tool DryadSynth-BV for bitvector synthesis [12], we do not include it in our comparison for the following reason. DryadSynth-BV employs various techniques to enhance enumeration, including term-graph-based enumeration, example-guided



(a) # Uniquely solved in the string domain



(b) # Uniquely solved in the bitvec domain

Benchmark category	# Solved			Time (Average)			Time (Median)		
	Hope	Duet	Synthphonia-G	H	D	S	H	D	S
Hard	**91**	23	42	13.01	2.0	30.82	4.56	0.06	40.88
Duet	**203**	186	202	1.11	0.38	6.71	0.03	0.02	0.22
Overall	**294**	209	244	4.79	0.56	10.86	0.06	0.02	0.56

(c) Statistics for the solving times for the string domain.

Benchmark category	# Solved			Time (Average)			Time (Median)		
	Hope	Duet	Simba	H	D	S	H	D	S
HD	**43**	35	42	2.85	2.95	2.36	0.12	0.13	0.16
Deobfusc	**462**	384	452	2.62	4.48	1.63	0.11	0.15	0.14
BitVec-Cond	**670**	619	533	12.68	10.18	6.23	1.10	2.92	1.47
Overall	**1175**	1038	1027	8.36	7.83	4.05	0.31	2.2	0.34

(d) Statistics for the solving times for the bitvector domain.

Fig. 7. Main result comparing the performance of Hope, Duet, Simba, and SynthPhonia-G on string and bitvector synthesis benchmarks.

filtration, and large language model guidance. It also utilizes a carefully designed enumeration order based on term graph size and syntactic ordering rules. These techniques could be integrated with our tool since they are complementary and orthogonal to our approach. However, since we do not have the integration in our prototypical implementation, we do not include DryadSynth-BV not to confuse the reader with the comparison of different aspects of synthesis techniques.

5.2 Comparison to Existing Synthesizers

We evaluate Hope on all the benchmarks and compare it against Duet, Simba, and SynthPhonia-G. For each instance, we measure the running time of synthesis using a timeout of 60 seconds.

The results are summarized in Fig. 7, which shows the number of problems solved by each synthesizer and the average solving time for each benchmark suite. Fig. 7c and Fig. 7d show the overall statistics for the string and bitvector domains, respectively. The average underlying-analysis times are 1.48s (string) and 5.43s (bitvector), while the average meta-analysis times are 1.38s (string) and 0.83s (bitvector). The average program sizes synthesized are 23.2 (string) and 107.1 (bitvector). Fig. 7a and Fig. 7b show the number of uniquely solved problems in the string and bitvector domains, respectively.

Overall, Hope outperforms the other baselines in both domains in terms of the number of problems solved. As shown in Fig. 7c, for the string domain, Hope solves **294** out of **337** string synthesis problems (87.2%) outperforming SynthPhonia-G which solves **244** problems (72.4%) and Duet which solves **209** problems (62.0%). For the bitvector domain, as shown in Fig. 7d, Hope

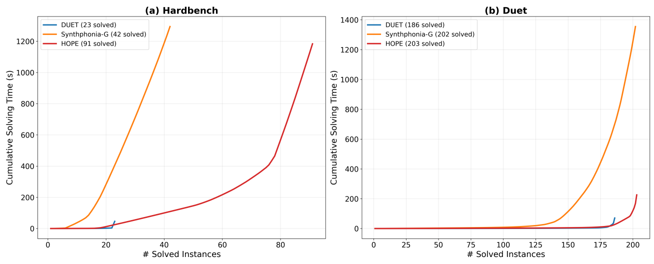


Fig. 8. Comparison between Hope and the other baseline solvers in the string domain.

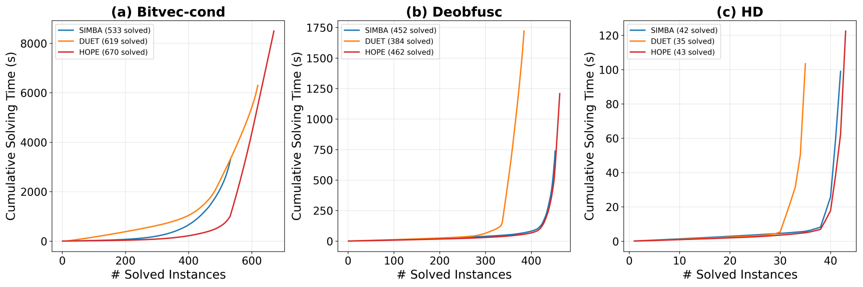


Fig. 9. Comparison between Hope and the other baseline solvers in the bitvector domain.

solves **1,175** out of **1,294** bitvector synthesis problems (90.8%) outperforming Simba which solves **1,098** problems (84.9%) and Duet which solves **1,014** problems (78.3%).

Hope solves hard problems that the other baselines cannot solve. As can be seen in Fig. 7a and Fig. 7b, in terms of the number of uniquely solved problems, Hope uniquely solves **47** problems in the string domain outperforming SynthPhonia-G which uniquely solves only **4** problems and Duet which uniquely solves no unique problems. In the bitvector domain, Hope uniquely solves **76** problems outperforming Simba which uniquely solves no problems and Duet which uniquely solves **51** problems.

Fig. 8 and Fig. 9 show the cactus plots for the string and bitvector domains, respectively. The cactus plots show the cumulative solving time of each synthesizer, with the x-axis representing the number of problems solved and the y-axis representing the cumulative solving time in seconds. As can be seen in the cactus plots, Hope consistently outperforms the other synthesizers in both domains, solving more problems within the time limit and with lower cumulative solving time.

In terms of average solving time, Hope is not the fastest one. This is because Hope solves more difficult problems than the other tools at the cost of longer average solving time and incurs some overhead due to the meta analysis.

Result in Detail. We study the results in detail for both domains, focusing on specific problem types and analyzing the performance of Hope compared to the baselines. Table 1 shows the detailed results on randomly chosen 15 problems (3 for each). The results indicate that Hope outperforms the other baselines in terms of the number of problems solved and the solving time across different problem types. The cost of meta analysis pays off by the fact that Hope is always faster than Simba, which can be regarded as Hope without meta analysis for the bitvector domain. The time spent for meta analysis($T_{A^\#}$) ranges from >0.01s to 7.9s, which is not significant compared to the overall solving time in most cases, demonstrating the efficiency of our lightweight meta analysis.

Table 1. Results for 15 randomly chosen benchmark problems (3 for each category), where **Time** gives synthesis time, T_A gives time spent for underlying analysis, $T_{A^\#}$ gives time spent for meta analysis, and $|P|$ shows the size of the synthesized program (measured by number of AST nodes).

Benchmark category	Benchmark	Simba		Duet		SP-G		Hope			
		Time	$\lvert P\rvert$	**Time**	$\lvert P\rvert$	**Time**	$\lvert P\rvert$	**Time**	T_A	$T_{A^\#}$	$\lvert P\rvert$
Hard	research_author4	-	-	**TO**		**TO**		9.29	3.16	0.02	134
	profiles_price1	-	-	**TO**		51.40	364	10.76	4.06	0.23	172
	research_year2	-	-	0.09	23	17.03	35	0.07	0.00	0.01	22
Duet	univ_1-long-repeat.sl	-	-	0.01	7	0.08	7	0.02	0.00	0.00	7
	stackoverflow1.sl	-	-	0.02	11	1.94	9	0.03	0.00	0.00	17
	univ_4_short.sl	-	-	**TO**		26.57	75	19.91	11.09	1.97	82
HD	hd-02-d0-prog	0.13	5	0.10	5	-	-	0.09	0.36	0.00	5
	hd-09-d0-prog	0.18	9	0.15	9	-	-	0.21	0.16	0.01	9
	hd-20-d0-prog	39.75	15	**TO**	-	-	-	24.12	6.01	2.90	15
Deobfusc	target_116	0.13	11	0.46	15	-	-	0.10	0.21	0.00	11
	target_57	0.12	11	1.50	9	-	-	0.09	0.29	0.00	11
	target_32	**TO**		**TO**	-	-	-	**TO**			
BitVec-Cond	75_1000	45.07	94	28.19	13	-	-	20.53	9.24	2.64	77
	82_100	**TO**		2.48	270	-	-	44.13	4.76	7.86	618
	51_10	0.26	11	1.75	11	-	-	0.17	0.20	0.00	11

Summary of Results. Hope solves a wide range of synthesis problems across both domains, demonstrating the effectiveness of its meta analysis-based hole-filling strategy.

5.3 Efficacy of Meta Analysis-Based Hole-Filling

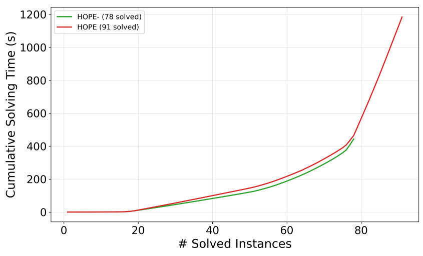
We now evaluate the effectiveness of the meta analysis-based hole-filling strategy in Hope. We compare the performance of Hope against its variant without the meta analysis-based hole-filling strategy, denoted as Hope$^-$. For the string domain, we only use the Hardbench benchmark for the ablation study, since there are no significant differences in performance on the Duet benchmark. For the bitvector domain, we use all benchmarks for the ablation study.

Fig. 10 shows the results of the ablation study. Table. 10c shows the overall statistics for the ablation study, and Fig. 10a and Fig. 10b show the cactus plots for the string and bitvector domains, respectively. In total, Hope solves 161 additional problems compared to Hope$^-$, demonstrating the effectiveness of the meta analysis-based hole-filling strategy.

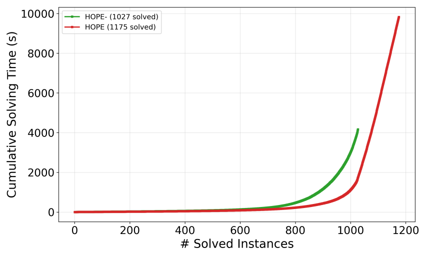
Summary of Ablation Study Results. The ablation study confirms that the meta analysis-based hole-filling strategy significantly improves the synthesis performance of Hope in both domains.

5.4 Discussion

Using a More Precise Abstract Domain for Meta Analysis. In our current implementation, we use a lightweight abstract domain for meta analysis to minimize overhead. To explore if there are potential benefits of a more precise abstract domain, we conducted preliminary experiments using a more precise abstract domain for meta analysis in the bitvector domain. We implemented a variant where the meta-analysis uses the same abstract domain as the underlying analysis. Specifically, each hole is hypothetically filled by abstracting the output values of all possible components into an element of the underlying abstract domain. The hole's rank is then computed by performing the underlying analysis with this hypothetical hole-filling. Because the meta-analysis domain and underlying analysis domain coincide, this variant represents the most precise meta-analysis possible. We evaluated it on 46 Hacker's Delight problems and found it solved 3 fewer problems and required 1 second longer on average than our original tool. The result suggests that while a more



(a) Hope and Hope^{-} for strings



(b) Hope and Hope^{-} for bitvectors

Benchmark category	# Solved		Time (Avg)		Time (Med)	
	Hope^{-}	Hope	Hope^{-}	Hope	Hope^{-}	Hope
Hard	78	**91**	5.68	13.01	3.74	4.56
All BitVec	1027	**1175**	4.05	8.36	0.34	0.31

(c) Statistics for the solving times for the ablation study.

Fig. 10. Result for the ablation study.

precise meta-analysis can in principle offer a better hole-filling strategy, the overhead outweighs the benefits in practice.

Impact of Multi-Hole Filling. As explained in §4.5, we fill multiple holes simultaneously to improve efficiency. To see how the maximum number of holes filled at a time affects performance, we have performed additional experiments for the string benchmarks. We have compared three configurations: (1) one hole at a time, (2) up to three holes at a time, and (3) up to five holes at a time. The best performance was achieved with up to three holes, solving 294 tasks in 4.79 seconds on average. Filling one hole at a time solved 251 tasks in 6.60 seconds, while filling up to five holes solved 292 tasks in 6.21 seconds. This demonstrates a clear trade-off: filling multiple holes helps by pruning infeasible candidates earlier. However, filling too many holes at once incurs combinatorial explosion in the space of hole-component combinations, leading to slower synthesis.

Case Study on Meta Analysis Guidance. To illustrate how meta analysis guides hole-filling order, we present a case study on the bitvector synthesis problem `hd-20-d5-prog` from the Hacker's Delight benchmark. This problem is difficult due to its large solution size (20 nodes in its syntax tree). The solution contains the subexpression $(x \oplus (-x \land x)) \gg 2$, which can be synthesized from the partial program $\square \gg \square$.

Simba fills the leftmost hole first, but the underlying analysis cannot infer any useful precondition for the remaining hole in this case. By contrast, our meta-analysis instead identifies that filling the second hole first is more promising: if the shift amount (the second hole) is instantiated to a constant, the underlying backward analysis can propagate a useful precondition back to the first hole (by shifting the desired output leftward by the chosen constant). This guidance enables more efficient synthesis of the full expression by pruning infeasible components for the first hole earlier. As a result, Hope solves this benchmark in 58.97 seconds, while Simba times out.

6 Related Work

Hole-Filling Strategies in Program Synthesis. Though deciding which hole to fill next during top-down synthesis significantly affects the search efficiency, the choice of hole-filling order is often overlooked in program synthesis literature. To the best of our knowledge, our work is the first to propose a generic hole-filling strategy that can improve the effect of an underlying pruning method in a synthesis engine. The most common approach is to fill holes in a fixed order, such as left-to-right [7, 20, 24, 38], depth-first [2, 23], or bottom-to-top [16]. This simple approach can lead to inefficient search as already shown in §2,§5. Neo [15] uses statistical models to predict the most promising hole to fill next. While this approach can adapt to the problem at hand, statistical models are not aware of the pruning method used in the synthesis engine, which can lead to suboptimal hole-filling orders in terms of pruning the search space. Flashmeta [30] leverages user-provided inverse semantics of operators usable in the synthesis task to determine the hole-filling order. Inverse semantics are functions that take a result and return a set of possible inputs that can produce the result, enabling an efficient top-down search. It extracts a dependency graph from the structure of inverse semantics It creates a directed acyclic graph over parameters of the operator, which FlashMeta uses to determine hole filling order via a topological sort. While this approach is effective, it requires the user to provide inverse semantics for each operator, which is not always feasible (not all operators have inverse semantics, and even if they do, it can be difficult to efficiently compute them). In contrast, thanks to the generality of abstract interpretation, our method is generally applicable to target languages possible with operators that do not have inverse semantics.

Abstract Interpretation in Program Synthesis. A number of approaches have used abstract interpretation to prune the search space in various domains [16, 20, 25, 32–37, 37, 38]. These approaches have focused only on overapproximating the behavior of candidate programs. In contrast, our approach overapproximates not only the behavior of candidate programs but also the behavior of the pruning method used in the synthesis engine to maximize its effect during the search.

Abstract^2 Interpretation. Abstract^2 interpretation [10] (shortly A^{2}I), also called meta-abstract interpretation, applies abstract interpretation to abstract interpretation-based program analyses. A^{2}I aims to overapproximate the sequence of fixpoint iterates of an abstract interpreter. It has been used to explain the prior work on, for example, improving the precision of widening [5, 22] and automatic variable packing in relational abstract domains [4, 17, 27]. Our approach also can be seen as a form of A^{2}I, where we overapproximate the final fixpoint of the underlying analysis rather than the sequence of fixpoint iterates.

Abstract Domains for Strings. There has been a number of abstract domains for strings, including domains using prefixes, suffixes, and character sets [8]. To the best of our knowledge, none of these domains are equipped with backward abstract operators, which are essential for our approach to work. There are also more complicated abstract domains for strings based on finite state automata [26], regular expressions [28], pushdown automata [21], parse stacks [14], and relational domains [1]. These domains are more precise than our domains, but their abstract operators are also more expensive to compute. Our domains are designed to be simple and efficient, while still being able to capture the essential properties of strings needed for program synthesis.

7 Conclusion

We have presented a novel hole-filling strategy for inductive program synthesis that leverages abstract interpretation to guide the order of hole-filling. Our approach uses a lightweight meta analysis that overapproximates the behavior of the underlying abstract interpreter for pruning, enabling it to predict the most promising hole to fill next. The experimental results show that our approach significantly outperforms the state-of-the-art approaches in terms of efficiency by pruning the search space early.

Acknowledgments

We thank the reviewers for their insightful comments that helped us improve the paper. This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (Nos. RS-2021-NR060080, 0536-20250107), the Institute for Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (Nos. 2022-0-00995, RS-2024-00341722, RS-2024-00423071), BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, Seoul National University) (4199990214639), Upstage Co., Ltd. (0536-20250102), and Greenlabs Co., Ltd. (0536-20220078).

References

[1] Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Pietro Ferrara. 2022. Relational String Abstract Domains. In *Verification, Model Checking, and Abstract Interpretation*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer International Publishing, Cham, 20–42.

[2] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

[3] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard Version 2.0.

[4] Julien Bertrane, Patrick Cousot, Radhia Cousot, JÃ©rÃŽme Feret, Laurent Mauborgne, Antoine MinÃ©, and Xavier Rival. 2015. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Foundations and TrendsÂ® in Programming Languages* 2, 2-3 (2015), 71–190. doi:10.1561/2500000002

[5] Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin. 1999. Polyhedral Analysis for Synchronous Languages. In *Static Analysis*, Agostino Cortesi and Gilberto Filé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–68.

[6] 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, Article 33 (Jan. 2023), 30 pages. doi:10.1145/3571226

[7] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction . In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 963–975. doi:10.1109/ASE56229.2023.00129

[8] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Formal Methods and Software Engineering*, Shengchao Qin and Zongyan Qiu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 505–521.

[9] Patrick Cousot and Rahida Cousot. 1992. Abstract Interpretation and Application to Logic Programs. *J. Log. Program.* 13, 2–3 (jul 1992), 103–179. doi:10.1016/0743-1066(92)90030-7

[10] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^{2}I: abstract^2 interpretation. *Proc. ACM Program. Lang.* 3, POPL, Article 42 (Jan. 2019), 31 pages. doi:10.1145/3290355

[11] Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Yuantian Ding and Xiaokang Qiu. 2024. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. *Proc. ACM Program. Lang.* 8, POPL, Article 71 (Jan. 2024), 31 pages. doi:10.1145/3632913

[13] Yuantian Ding and Xiaokang Qiu. 2025. A Concurrent Approach to String Transformation Synthesis. *Proc. ACM Program. Lang.* 9, PLDI, Article 233 (June 2025), 25 pages. doi:10.1145/3729336

[14] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. 2009. Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology. In *Proceedings of the 16th International Symposium on Static Analysis* (Los Angeles, CA) *(SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 256–272. doi:10.1007/978-3-642-03237-0_18

[15] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. doi:10.1145/3192366.3192382

[16] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.* 52, 6 (jun 2017), 422–436. doi:10.1145/3140587.3062351

[17] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–256.

[18] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 335–352.

[19] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. doi:10.1145/3385412.3385979

[20] Keith J.C. Johnson, Rahul Krishnan, Thomas Reps, and Loris D'Antoni. 2024. Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 304 (Oct. 2024), 27 pages. doi:10.1145/3689744

[21] Se-Won Kim and Kwang-Moo Choe. 2011. String Analysis as an Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 294–308.

[22] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. 2011. Widening with Thresholds for Programs with Complex Control Graphs. In *Automated Technology for Verification and Analysis*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 492–502.

[23] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[24] 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* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. doi:10.1145/3192366.3192410

[25] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-Based Pruning for Speeding up Superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 177 (nov 2020), 24 pages. doi:10.1145/3428245

[26] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2020. Twinning automata and regular expressions for string static analysis. arXiv:2006.02715 [cs.SE] https://arxiv.org/abs/2006.02715

[27] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2015. Selective X-Sensitive Analysis Guided by Impact Pre-Analysis. *ACM Trans. Program. Lang. Syst.* 38, 2, Article 6 (Dec. 2015), 45 pages. doi:10.1145/2821504

[28] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS 2016)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/2989225.2989228

[29] Past SyGuS Competition. 2020. https://sygus.org/comp/.

[30] 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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. doi:10.1145/2814270.2814310

[31] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE Press, 404–415. doi:10.1109/ICSE.2017.44

[32] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 289–299. doi:10.1145/2025113.2025153

[33] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381.

[34] Ma[rtin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-Guided Synthesis of Synchronization. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 327–338. doi:10.1145/1706299.1706338

[35] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. doi:10.1145/3062341.3062365

[36] Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426.

[37] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. doi:10.1145/3158151

[38] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (June 2023), 25 pages. doi:10.1145/3591288

Received 2025-07-10; accepted 2025-11-06