Presynthesis

Towards Scaling Up Program Synthesis with Finer-Grained Abstract Semantics

RUI DONG, University of Michigan, USA
QINGYUE WU, University of Michigan, USA
DANNY DING*, University of Texas at Austin, USA
ZHENG GUO, University of Michigan, USA
RUYI JI, University of Michigan, USA
XINYU WANG, University of Michigan, USA

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

Additional Key Words and Phrases: Program Synthesis, Automata Theory, Abstract Interpretation

ACM Reference Format:
Rui Dong, Qingyue Wu, Danny Ding, Zheng Guo, Ruyi Ji, and Xinyu Wang. 2026. Presynthesis: Towards Scaling Up Program Synthesis with Finer-Grained Abstract Semantics. *Proc. ACM Program. Lang.* 10, PLDI, Article 210 (June 2026), 25 pages. https://doi.org/10.1145/3808288

Abstract semantics has proven to be instrumental for accelerating search-based program synthesis, by enabling the sound pruning of a set of incorrect programs (without enumerating them). One may expect *faster* synthesis with increasingly *finer-grained* abstract semantics. Unfortunately, to the best of our knowledge, this is not the case, yet. The reason is because, as abstraction granularity increases—while fewer programs are enumerated—pruning becomes more costly. This imposes a fundamental limit on the overall synthesis performance, which we aim to address in this work.

Our key idea is to introduce an *offline presynthesis* phase, which consists of two steps. Given a DSL with abstract semantics, the first *semantics modeling* step constructs a tree automaton $\mathcal{A}$ for *a space of inputs*—such that, for any program P and for any considered input I, $\mathcal{A}$ has a run that corresponds to P's execution on I under abstract semantics. Then, the second step builds an oracle O for $\mathcal{A}$. This O enables fast pruning *during synthesis*, by allowing us to efficiently find exactly those DSL programs that satisfy a given input-output example under abstract semantics.

We have implemented this *presynthesis-based synthesis paradigm* in a framework, Foresighter. On top of it, we have developed three instantiations for SQL, string transformation, and matrix manipulation. All of them significantly outperform prior work in the respective domains.

*Work done while working as an undergraduate researcher at the University of Michigan, Ann Arbor.

Authors' Contact Information: Rui Dong, University of Michigan, Ann Arbor, USA, ruidong@umich.edu; Qingyue Wu, University of Michigan, Ann Arbor, USA, wqingyue@umich.edu; Danny Ding, University of Texas at Austin, Austin, USA, dingyy@utexas.edu; Zheng Guo, University of Michigan, Ann Arbor, USA, zhgguo@umich.edu; Ruyi Ji, University of Michigan, Ann Arbor, USA, jiry@umich.edu; Xinyu Wang, University of Michigan, Ann Arbor, USA, xwangsd@umich.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART210
https://doi.org/10.1145/3808288

1 Introduction

This paper concerns the long-standing problem of syntax-guided program synthesis from examples, also known as programming-by-example (PBE). That is,

> Given a domain-specific language (DSL) whose syntax is defined by a context-free grammar and given a set E of input-output examples, find a program P in the DSL such that P satisfies E according to the DSL's semantics.

This is an intellectually challenging problem with a wide range of practical applications, evidenced by the seminal FlashFill work [18] among many others [5, 19, 41, 45–47, 56]. A widely-used synthesis method is to *search* over the given grammar for satisfying programs. But, as one can imagine, this is computationally intractable, due to the enormous search space of programs.

State-of-the-art. To scale up synthesis, many state-of-the-art synthesizers leverage some form of *abstract semantics* to prune the search space [15, 17, 21, 29, 40, 53, 56]. The idea is to reason about the behavior of *a set of programs* to prove their inconsistency with the specification, such that they can be safely pruned away. For example, top-down synthesis [15] works by enumerating program sketches (i.e., incomplete programs with holes). A sketch can be pruned away if its abstract behavior is not consistent with the example, thereby saving us from enumerating any of its completions. As another example, in representation-based synthesis [56], a graph-like data structure is first built for the given input-output example, based on the DSL's abstract semantics. In this graph, nodes are associated with abstract output values of programs on the given input, edges are created using abstract semantics, and paths represent programs. Then, synthesis boils down to searching among "consistent" paths (i.e., paths that reach a target node whose associated abstract value is consistent with the desired output). Irrelevant nodes (that do not lie on a consistent path) are not considered during search; in other words, their corresponding programs are pruned away.

Finer-grained abstraction ⇒ fast synthesis? The idea of abstraction-based pruning is profound, and underpins search-based synthesis techniques across a wide range of applications. In this work, we ask the following question.

Can we scale up synthesis by providing it with increasingly finer-grained abstract semantics?

This is an important question, because a positive answer indicates that we can potentially use abstraction granularity as a dimension to continuously scale up synthesis. Unfortunately, to our best knowledge, this is not the case, yet. While developing precise abstract semantics is itself very hard [25, 55], an orthogonal challenge is that *pruning is not always cheap.* With a finer abstraction, while more programs can potentially be pruned away, the cost of pruning also increases. As the abstraction becomes more granular, this overhead can offset—or even outweigh—its benefit to synthesis. For instance, in top-down synthesis, evaluating a program sketch using a finer-grained abstraction is often more costly [15], while potentially still failing to prune it away. In representation-based synthesis, finer-grained abstract semantics will result in a larger graph *with more irrelevant nodes* [56]. This "graph bloat" phenomenon is fundamentally because we cannot soundly predict, during its construction, which part of the graph is relevant, until the graph is fully built.

Our work: presynthesis-based synthesis. We present a scalable pruning technique, which we believe is a crucial step towards scaling up synthesis with finer-grained abstract semantics. Figure 1 shows the schematic workflow. The key idea is to introduce an *offline presynthesis* phase. Given a DSL with abstract semantics, we *precompute* some information, which can be stored on a disk. To solve a stream of synthesis tasks, such information is first loaded into the synthesizer, and will

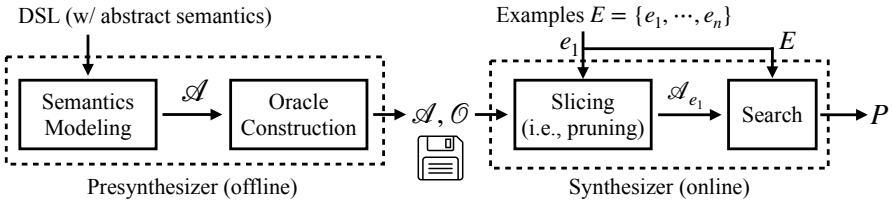


Fig. 1. Schematic workflow of our presynthesis-based synthesis paradigm.

be used to enable fast pruning. This is a paradigm shift, from existing work[1] which is virtually all "purely online", to one that can take advantage of *offline* resources for better *online* performance. Two key ideas allow us to realize this "presynthesis-based synthesis" paradigm.

Key idea 1: offline abstract semantics modeling using FTAs. Our first idea is to model the given DSL's abstract semantics for *a space of inputs* (that are of interest for future synthesis). Specifically, given a DSL with abstract semantics, we build a finite tree automaton (FTA) $\mathcal{A}$ during presynthesis, such that, for any program P and for any input in the considered space, $\mathcal{A}$ has a run ρ that begins from an abstraction of the given input and ends in a state labeled with P's resulting abstract output.

Key idea 2: fast pruning using offline-built oracle. Given this offline FTA $\mathcal{A}$, synthesis boils down to finding a run ρ of $\mathcal{A}$ that is *consistent with* the given input-output example $e = e_{in} \mapsto e_{out}$. Here, ρ is said to be consistent with e, if it (i) begins from states consistent with e_{in} and (ii) ends in a state that matches e_{out}. This can be viewed as a "graph search" problem; the challenge lies in that our $\mathcal{A}$ is very large. Our idea is to first build a form of "reachability oracle" $\mathcal{O}$ *during presynthesis*. Then, *during synthesis*, $\mathcal{O}$ is used to help extract a *slice* $\mathcal{A}_e$ (which is a sub-FTA of $\mathcal{A}$) for e, such that $\mathcal{A}_e$ represents all programs that satisfy e under abstract semantics. This slicing step essentially performs pruning. However, unlike prior work, our slicing-based pruning is very fast, thanks to our oracle $\mathcal{O}$ (which significantly reduces the "graph bloat" while building $\mathcal{A}_e$). Finally, we perform standard search over $\mathcal{A}_e$ to find a program that satisfies all examples under concrete semantics.

Synthesis as a service. As one can imagine, in order for the synthesizer to fully load in the offline artifact, more memory is needed (compared to purely online approaches). Therefore, in general, we envision our presynthesis-based synthesizer to be deployed as a *service*: that is, user requests (i.e., specifications) are sent to a *synthesis server*, and the synthesized programs are sent back to users. Note that users do not need as much memory, unless they choose to deploy the service locally.

Implementation, instantiation, and evaluation. We have implemented this presynthesis-based synthesis paradigm into a framework called Foresighter, and instantiated it for three different domains. In addition to outperforming prior work across all domains, it is also worth highlighting that, as the abstraction granularity increases (up to the finest available), the pruning time of our approach stays consistently low and the synthesis time continues to decrease.

Contributions. This paper makes the following contributions.

- The concept of presynthesis, and a new presynthesis-based synthesis paradigm.
- The first presynthesis technique.
- The first synthesis algorithm that utilizes presynthesis-based pruning.
- Implementation of the presynthesis-based synthesis framework, and three instantiations.
- Comprehensive evaluation, with promising experimental results across the board.

[1]We refer interested readers to Section 6 for related work that utilizes preprocessing/precomputation.

2 Overview

This section illustrates our approach over a simple synthesis task.

Domain-specific language (DSL). Consider the following unrolled grammar[2], which specifies the syntax of a small bit-vector language from the SyGuS dataset [4, 5]. For presentation purposes, we consider vectors of length 4, left shift $\ll$ and addition +, one single input variable x, and two integer constants for k. We further assume only t_2 is the start symbol. The semantics is standard.

$$\begin{array}{lcl} t & ::= & x \mid t + k \mid t \ll k \\ k & ::= & 1 \mid 2 \end{array} \xrightarrow{\text{Unrolling}} \begin{array}{lcl} t_2 & ::= & t_1 + k \mid t_1 \ll k \\ t_1 & ::= & t_0 + k \mid t_0 \ll k \\ t_0 & ::= & x \\ k & ::= & 1 \mid 2 \end{array}$$

Programming-by-example (PBE) task. Given this DSL, consider a PBE task with the following set E of two examples. The PBE task is to find a DSL program that satisfies both examples in E.

$$E = \left\{ \begin{array}{l} e_1 : \{\, x = 0001 \,\} \mapsto 0110 \\ e_2 : \{\, x = 0010 \,\} \mapsto 1010 \end{array} \right\}$$

In our unrolled grammar, $(x \ll 2) + 2$ is the only program that satisfies E.

2.1 FTA-Based Synthesis using Abstract Semantics

Let us first illustrate an existing PBE framework [54, 56]—which our work builds upon—that is based on finite tree automata (FTAs) and abstract semantics.

Abstract semantics. The abstract semantics for our bit-vector DSL is defined by a set of abstract transformers for all operators in the DSL. A transformer for operator f accepts abstract values as arguments and returns an abstract value as its output. Abstract values are drawn from a predefined domain of abstract values (i.e., abstract domain). Below are some example transformers.

$$[\![\square 1 \ll 1]\!]^\sharp = \square 0 \qquad [\![\square 1 \ll 2]\!]^\sharp = \square 0 \qquad [\![\square 1 + 1]\!]^\sharp = \square 0 \qquad [\![\square 1 + 2]\!]^\sharp = \square 1$$

$\square b$ denotes an (abstract) vector whose lowest bit is b. For instance, the first example says "performing left shift on a vector whose lowest bit is 1 always yields a vector whose lowest bit is 0".

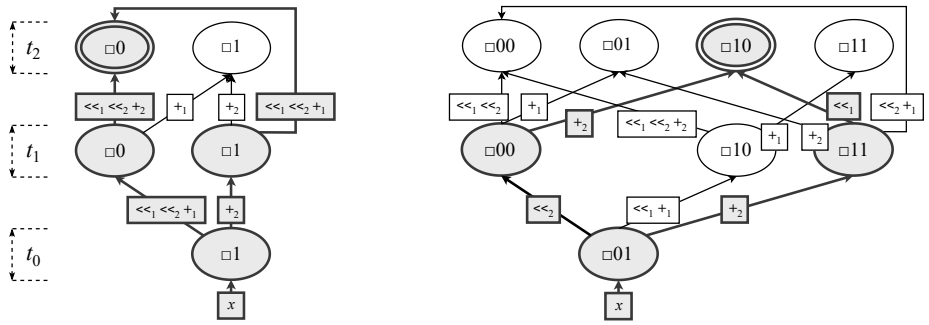
PBE using FTAs and abstract semantics. We can perform synthesis by first building an FTA, given a DSL with abstract semantics and given an example. Figure 2a shows the FTA $\mathcal{A}_1$ for the first example e_1, given our DSL with the aforementioned abstract semantics. Each state q_s^a is associated with a grammar symbol s and an abstract value a. Each transition corresponds to a DSL operator f, and is built using f's abstract transformer $[\![f]\!]^\sharp$. An accepting run ρ of $\mathcal{A}_1$ represents a program that satisfies e_1 under abstract semantics. For instance, consider the following run ρ of $\mathcal{A}_1$

$$\xrightarrow{x} q_{t_0}^{\square 1} \xrightarrow{\ll_1} q_{t_1}^{\square 0} \xrightarrow{+_2} q_{t_2}^{\square 0}$$

which corresponds to $(x \ll 1) + 2$. This program, however, does not satisfy e_1 under the DSL's *concrete* semantics. It is not hard to figure out that $\mathcal{A}_1$ has in total 12 accepting runs, while the DSL has a total of 16 programs. Given $\mathcal{A}_1$, we can perform search *among its accepting runs*, to find a program that satisfies both e_1 and e_2 under concrete semantics.

Remarks. While $\mathcal{A}_1$ prunes away 4 programs (which are not considered during search), they are still presented in Figure 2a as runs ending at $q_{t_2}^{\square 1}$. This is because states are built in a bottom-up fashion: begin with states for input variables, iteratively create more states, until all programs in the DSL are considered. For instance, in Figure 2a, transition $\xrightarrow{x} q_{t_0}^{\square 1}$ is first created, since e_1 binds

[2]We assume an unrolled grammar to simplify the presentation.



(a) FTA $\mathcal{A}_1$ for e_1, tracking lowest bit.

(b) FTA $\mathcal{A}_2$ for e_1, tracking lowest two bits.

Fig. 2. Two FTAs for input-output example $e_1 : \{x = 0001\} \mapsto 0110$. $\mathcal{A}_1$ in Figure 2a uses an abstract domain that tracks the lowest bit for all bitvectors, and $\mathcal{A}_2$ in Figure 2b uses a finer-grained domain that tracks the lowest two bits. Each state is labeled with its associated abstract value, and the associated grammar symbol is shown on the far left. For presentation purposes, transitions for $\ll$ and + also include constant k as part of the label (boxed), and we merge labels for transitions that share the same input and output states. Final states are double-circled. Accepting runs are highlighted with gray states and bold transitions.

x to 0001 whose lowest bit is 1. Note the state $q_{t_2}^{\square 1}$: while "useless", it is still created—this is because, in general, we cannot foresee whether or not a state can lead to a final state. As a result, prior work first builds all "reachable" states from the input example and then removes those not reaching a final state (i.e., the desired output example).

2.2 The FTA Bloat Phenomenon

Now let us illustrate the FTA bloat issue when a finer-grained abstract semantics is used.

Finer-grained abstract semantics. Consider a more precise abstract domain, which tracks the lowest two bits. The abstract transformers can be defined accordingly. Below are some examples.

$$[\![\square 01 \ll 1]\!]^\sharp = \square 10 \qquad [\![\square 11 \ll 2]\!]^\sharp = \square 00 \qquad [\![\square 01 + 1]\!]^\sharp = \square 10 \qquad [\![\square 00 + 2]\!]^\sharp = \square 10$$

Fewer accepting runs. Figure 2b shows the FTA $\mathcal{A}_2$ for this new abstraction. Compared to $\mathcal{A}_1$ in Figure 2a, $\mathcal{A}_2$ is larger, since it tracks finer-grained information. As a result, $\mathcal{A}_2$ has fewer accepting runs—in particular, only 2, compared to 12 in $\mathcal{A}_1$. In other words, more programs are pruned away. This makes the subsequent search faster.

More bloat. However, this additional pruning power and faster search do not come for free—$\mathcal{A}_2$ has more "useless" states and transitions. As alluded to earlier, this FTA bloat phenomenon is due to the inability to soundly "predict the future" (i.e., if a current state will lead to a future final state). For instance, $q_{t_1}^{\square 10}$ in $\mathcal{A}_2$ does not lead to a final state, no matter what operators and constants are used; but, in general, we must first build all of its outgoing transitions and states, to soundly tell so. This FTA bloat issue is exacerbated when finer-grained abstract semantics is used.

2.3 Our Approach: Presynthesis-Based Synthesis

Finally, let us illustrate how we enable fast pruning without causing FTA bloat during synthesis.

Offline FTA-based semantics modeling. Our FTA is built during a *presynthesis* phase. Figure 3 shows our offline FTA $\mathcal{A}$, built using the same abstraction for $\mathcal{A}_2$. Unlike $\mathcal{A}_2$, $\mathcal{A}$ has 3 additional

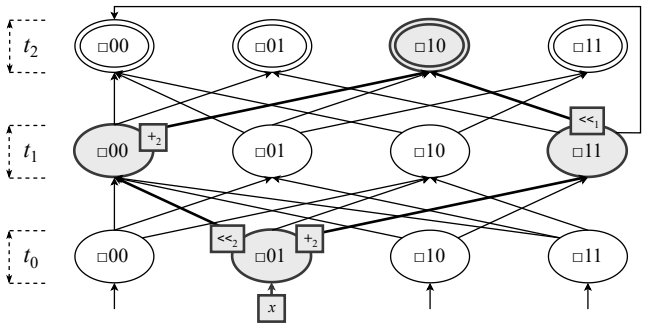


Fig. 3. Offline FTA $\mathcal{A}$ for our DSL using abstract semantics that tracks the lowest two bits. We highlight the part of $\mathcal{A}$ that corresponds to the slice $\mathcal{A}_{e_1}$ for the first example $e_1 : \{x = 0001\} \mapsto 0110$, and we elide labels for transitions not in $\mathcal{A}_{e_1}$ to simplify the presentation.

"initial" states for x, because $\mathcal{A}$ needs to consider all potential input vectors. Another distinction is that all t_2 states are final, because any of them can be desired before "online examples" are revealed. The rest of the FTA construction machinery remains the same.

Our new insight is that $\mathcal{A}$ essentially models the DSL's abstract semantics—in the sense that, the execution of any program P on any input corresponds to an accepting run ρ of $\mathcal{A}$. Prior work that builds FTAs *online* also performs such modeling, but only for specific inputs.

Online slicing. Now, consider the synthesis task for our examples $E = \{e_1, e_2\}$. First, we extract a *slice* $\mathcal{A}_{e_1}$ out of $\mathcal{A}$. $\mathcal{A}_{e_1}$ is a sub-FTA of $\mathcal{A}$, as highlighted in Figure 3. $\mathcal{A}_{e_1}$'s accepting runs represent exactly those programs that satisfy e_1 abstractly. Readers now might wonder: how is $\mathcal{A}_{e_1}$ different from $\mathcal{A}_2$ (see Figure 2b)? Indeed, they have the language. The key distinction is that our slicing does not suffer from the bloat issue as when building $\mathcal{A}_2$. Let us illustrate our slicing process below.

Given $\mathcal{A}$ and e_1, we first find final states that are consistent with the output example: in our case, $q_{t_2}^{\square 10}$ is the only such state. Then, we traverse $\mathcal{A}$ top-down from $q_{t_2}^{\square 10}$ to collect its "descendants". For instance, we will collect $q_{t_2}^{\square 00}$ and $q_{t_2}^{\square 11}$, both of which belong to $\mathcal{A}_{e_1}$, as well as $q_{t_0}^{\square 01}$. However, we will not collect $q_{t_1}^{\square 01}$, because it is reachable only from initial states that are *not consistent* with e_1's input. Similarly, we will not collect $q_{t_0}^{\square 00}$ and $q_{t_0}^{\square 11}$ either. In other words, we smartly avoid *irrelevant* states that end up not in $\mathcal{A}_{e_1}$ (which is the root cause of bloat!). How can we foresee the future?

Offline-built oracle. Our foresight comes from the use of an oracle O, that is built offline during presynthesis. Intuitively, O tells us whether or not a state q in $\mathcal{A}$ is reachable from initial states that are consistent with e_1's input: if so, q is relevant, and will be added to our slice $\mathcal{A}_{e_1}$. In particular, O maps every q to all *abstract inputs* that "can reach" q. For instance, O contains the following entry

$$O(q_{t_1}^{\square 01}) \quad = \quad \big(x = \square 00\big) \vee \big(x = \square 11\big)$$

which means: there is a run to $q_{t_1}^{\square 01}$ in $\mathcal{A}$, only if x binds to □00 or □11. Now, consider our example e_1 where the input binds x to the concrete value 0001. The reason that, during the previous top-down traversal process, we did not collect $q_{t_1}^{\square 01}$ is because: e_1's input 0001 is not consistent with any of the abstract values that O maps $q_{t_1}^{\square 01}$ to. On the other hand, we have

$$O(q_{t_1}^{\square 00}) = \big(x = \square 00\big) \vee \big(x = \square 01\big) \vee \big(x = \square 10\big) \vee \big(x = \square 11\big) \quad O(q_{t_1}^{\square 11}) = \big(x = \square 01\big) \vee \big(x = \square 10\big)$$

both of which contain a consistent abstract input. That is why both states were collected. Note that we check O for final states too: in this case, O maps $q_{t_2}^{\square 10}$ to an abstract input consistent with e_1.

O is essentially a form of *reachability oracle* [9, 49], which is designed for our synthesis setting. It is built by processing $\mathcal{A}$ in a bottom-up fashion, starting from t_0 (i.e., initial) states.

$$O(q_{t_0}^{\square 00}) = (x = \square 00) \quad O(q_{t_0}^{\square 01}) = (x = \square 01) \quad O(q_{t_0}^{\square 10}) = (x = \square 10) \quad O(q_{t_0}^{\square 11}) = (x = \square 11)$$

Then, we build entries for each t_1 state, by combining its incoming transitions disjunctively. See $O(q_{t_1}^{\square 00}), O(q_{t_1}^{\square 01}), O(q_{t_1}^{\square 11})$ above as examples. Finally, t_2 states are processed in the same way.

Online search. Our slice $\mathcal{A}_{e_1}$ is essentially a grammar (with all programs that satisfy e_1 abstractly). Now, given $E = \{e_1, e_2\}$, we can in principle use any syntax-guided solver to search over $\mathcal{A}_{e_1}$ for programs that concretely satisfy E. Hence, our framework is not tied to a specific search algorithm.

3 Presynthesis-Based Synthesis Framework

3.1 Preliminaries

We first cover necessary formal background on a widely-adopted framework [56, 57] for syntax-guided synthesis from examples, that is based on finite tree automata (FTAs) and abstract semantics.

Domain-specific language (DSL). Following the seminal work on syntax-guided synthesis [5], we assume the DSL's syntax is defined by a context-free grammar $G = (V, \Sigma, R, s_0)$, where

- V is a finite set of non-terminal symbols, that correspond to (sub-)programs in the DSL.
- Σ is a finite set of terminal symbols, which correspond to DSL operators, input variables, and constants. We use $\Sigma_{var} \subseteq \Sigma$ to denote the set of all input variables.
- R is a set of production rules of the form $s \to f(s_1, \cdots, s_n)$, where $s, s_1, \cdots, s_n$ are non-terminals and f is an n-arity DSL operator. Input variables and constants are viewed as nullary operators.
- $s_0 \in V$ is the start symbol.

As standard in the literature [41, 45, 56], we finitize the grammar (e.g., by finite unrolling).

The DSL's (concretize) semantics is assigned by defining, for each DSL operator f, a concrete transformer $[\![f]\!]$, which computes the concrete output of f given concrete argument values. An input variable is evaluated given a binding environment in the usual way.

Programming-by-example (PBE). The PBE problem is to find a program P, from a given DSL, that satisfies a given set E of input-output examples according to the DSL's concrete semantics. Each example $e \in E$ is of the form $e_{in} \mapsto e_{out}$, where e_{in} is a binding environment that maps input variables to their values and e_{out} is the desired output for e_{in}.

Abstract semantics. Following the seminal work on abstract interpretation [11], we define the DSL's abstract semantics using abstract values and abstract transformers. In particular, for each production rule $s \to f(s_1, \cdots, s_n)$, its abstract transformer $[\![f]\!]^\sharp$ returns an abstract value a for s given abstract values $a_1, \cdots, a_n$ for f's argument symbols $s_1, \cdots, s_n$. Following prior work [56], we require such transformers to be sound; that is,

$$\text{if } [\![f]\!]^\sharp(a_1, \cdots, a_n) = a \text{ and } c_1 \in \gamma(a_1), \cdots, c_n \in \gamma(a_n), \text{ then } [\![f]\!](c_1, \cdots, c_n) \in \gamma(a).$$

Here, γ is the concretization function that maps an abstract value to a set of concrete values. We also require an abstraction function α that maps a concrete value to an abstract value—which will be used to perform abstraction-based synthesis, as will be shown shortly.

Finite tree automata (FTAs). FTAs [10] accept trees, which our work will utilize to represent a set of abstract syntax trees (ASTs) for DSL programs. More formally, a (bottom-up) FTA is a tuple $\mathcal{A} = (Q, F, Q_f, \Delta)$, where Q is a set of states, F is a (ranked) alphabet, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transitions of the form $f(q_1, \cdots, q_n) \to q$ with $f \in F$ and $q_1, \cdots, q_n, q \in Q$.

$$[\textsc{Syn-Var}]\ \frac{x \in \Sigma_{var} \qquad s \to x \in R \qquad a = \alpha^{\mathcal{D}_x}\left(e_{in}(x)\right)}{q_s^a \in Q \quad x \in F \quad x \to q_s^a \in \Delta} \qquad [\textsc{Syn-Final}]\ \frac{q_{s_0}^a \in Q \qquad e_{out} \in \gamma(a)}{q_{s_0}^a \in Q_f}$$

$$[\textsc{Syn-Prod}]\ \frac{s \to f(s_1, \cdots, s_n) \in R \qquad q_{s_i}^{a_i} \in Q \qquad a = [\![f]\!]^\sharp(a_1, \cdots, a_n)}{q_s^a \in Q \quad f \in F \quad f(q_{s_1}^{a_1}, \cdots, q_{s_n}^{a_n}) \to q_s^a \in \Delta}$$

Fig. 4. Rules for constructing $\mathcal{A} = (Q, F, Q_f, \Delta)$, given (i) a DSL whose grammar is $G = (V, \Sigma, s_0, R)$ and whose abstract semantics is defined by abstract transformers $[\![\cdot]\!]^\sharp$ and (ii) an input-output example $e = e_{in} \mapsto e_{out}$. $\alpha^{\mathcal{D}_x}$ is the abstraction function over the abstract domain of variable x, while γ is the concretization function.

$\mathcal{A}$ *accepts* a term t—which corresponds to a DSL program—if, using $\mathcal{A}$'s transitions (i.e., rewrite rules), t can be rewritten to one of $\mathcal{A}$'s final states. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of terms accepted by $\mathcal{A}$. Given any $t \in \mathcal{L}(\mathcal{A})$, an *accepting run* for t maps each sub-term (which corresponds to a sub-program) of t to a state in $\mathcal{A}$, and maps t to a final state.

Program synthesis using FTAs and abstract semantics. A recent line of work [54] developed a technique to construct an FTA $\mathcal{A}$ for a given DSL (with a given abstract semantics), such that $\mathcal{L}(\mathcal{A})$ is exactly those programs that satisfy a given input-output example e with respect to the DSL's abstract semantics. $\mathcal{A}$ can then be used for PBE [56–59]. Our work builds upon this approach. Figure 4 gives their FTA construction rules, which are briefly explained below.

In a nutshell, a state q_s^a is constructed in FTA $\mathcal{A}$, if a (sub-)program at grammar symbol s can produce abstract value a. A transition connects argument states to a return state based on abstract semantics. Specifically, the first rule, Syn-Var, creates transitions for input variables, where x's abstract value a is obtained by applying α to the concrete value that e_{in} binds x to. The Syn-Prod rule grows $\mathcal{A}$ by processing every production rule. Finally, Syn-Final marks a state $q_{s_0}^a$ final, if a is consistent with e_{out}. The soundness of abstract transformers ensures $\mathcal{A}$ is also sound—that is, every accepting run of $\mathcal{A}$ corresponds to a DSL program that satisfies e under the given abstraction. Since rules in Figure 4 are applied until saturation, $\mathcal{L}(\mathcal{A})$ has all programs that abstractly satisfy e. Hence, we can safely prune out programs not in $\mathcal{A}$, without hindering synthesis completeness.

Remarks. This FTA $\mathcal{A}$ essentially models the abstract semantics for the given example, by mapping all programs in the DSL to runs in $\mathcal{A}$. However, only those *accepting* runs represent abstractly valid programs, while the rest cause the FTA bloat issue. Our work addresses exactly this bloat.

3.2 Top-Level Algorithm

Algorithm 1 gives our top-level algorithm with two procedures, Presynthesize and Synthesize.

Presynthesis (offline phase). Presynthesize is performed offline—*once* for a given DSL. It yields an FTA $\mathcal{A}$ and its corresponding oracle $\mathcal{O}$. $\mathcal{A}$ models the abstract semantics for all DSL programs, for *a space of inputs* (which are of interest for future synthesis tasks). That is, given a program P, for any input from the considered space, $\mathcal{A}$ has a corresponding run for P which reveals P's output (abstract) value as well as all intermediate ones. This is different from all prior work (to our best knowledge), which builds an FTA *only* for those inputs specific to the synthesis task at hand. As a result, our offline $\mathcal{A}$ is expected to be larger than its online counterpart from prior work, but $\mathcal{A}$ is built during presynthesis. $\mathcal{O}$ is a data structure, also built offline, that is used to accelerate online synthesis, which we will explain shortly.

Synthesis (online phase). Before serving any synthesis requests, $\mathcal{A}$ and $\mathcal{O}$ are first loaded into a synthesis server. Then, given a synthesis task with a set E of examples, Synthesize will perform

Algorithm 1 Top-level algorithm for presynthesis-based synthesis.

procedure PRESYNTHESIZE(G)
input: G is a context-free grammar that is associated with abstract semantics.
output: An offline FTA $\mathcal{A}$ and its oracle O.
1: $\mathcal{A}$:= BUILDOFFLINEFTA(G);
2: O := BUILDORACLE($\mathcal{A}$);
3: **return** ($\mathcal{A}, O$);

procedure SYNTHESIZE($E, \mathcal{A}, O$)
input: $E = \{e_1, \cdots, e_n\}$ is a set of n input-output examples.
input: $\mathcal{A}$ and O are the FTA and oracle build by presynthesis.
output: A program P that satisfies E under concrete semantics, or *null* indicating no such program is found.
4: $\mathcal{A}_{e_1}$:= SLICE($\mathcal{A}, e_1, O$);
5: P := SEARCH($\mathcal{A}_{e_1}, E$);
6: **return** P;

SEARCH (as in prior work) to find a program that satisfies E (with respect to the concrete semantics). This SEARCH, however, is performed over a *slice* $\mathcal{A}_{e_1}$ that is built for one example $e_1 \in E$. $\mathcal{A}_{e_1}$ is a sub-FTA of our offline $\mathcal{A}$, and $\mathcal{L}(\mathcal{A}_{e_1})$ is all programs that satisfy e_1 under abstract semantics. $\mathcal{A}_{e_1}$ is computed using a new SLICE algorithm that leverages our oracle O to address the FTA bloat.

Addressing FTA bloat. Prior work can be viewed as also building a slice/FTA that represents all abstractly consistent programs. But, as illustrated in Section 2, they do it online (i.e., during synthesis), and suffer from the FTA bloat issue (i.e., creating too many unnecessary states/transitions). Note that the bloat cannot be addressed by only building an offline $\mathcal{A}$. We must also know how to navigate in $\mathcal{A}$, to avoid adding unnecessary states to the slice. Our oracle O is designed exactly for this purpose: our slicing algorithm, SLICE, utilizes O to compute the slice highly efficiently.

In what follows, we first describe how to build offline $\mathcal{A}$, and then present how slicing works. Since SEARCH can be done using any syntax-guided search technique, we skip its presentation.

3.3 Offline FTA Construction

Figure 5 gives the construction rules for BUILDOFFLINEFTA. It shares the same structure as Figure 4, except for two important differences. First, PRESYN-VAR creates transitions for x, for all abstract values in $\mathcal{D}_x$ that x can bind to. This is because we consider a space of inputs during presynthesis. Second, PRESYN-FINAL marks states associated with the grammar's start symbol s_0 as final states, regardless of the abstract value. This is because presynthesis does not have access to online examples yet. On the other hand, PRESYN-PROD remains the same as its counterpart in Figure 4.

In what follows, we highlight a few important characteristics of this offline FTA $\mathcal{A}$.

Non-deterministic FTA to model semantics for all inputs. The PRESYN-VAR rule—which builds *multiple* transitions for the same variable x—makes $\mathcal{A}$ non-deterministic. That is, one program now corresponds to multiple runs in $\mathcal{A}$. This non-determinism is intended, since $\mathcal{A}$ aims to model the abstract semantics *for all inputs* of interest, rather than for a specific one as in prior work.

Compactness by sharing states across both programs and inputs. $\mathcal{A}$ is compact in two ways. First, as in prior work, *for the same input*, different (sub-)programs are merged into one state if they return the same output. Second, (sub-)programs that yield the same output, on *different inputs*, are also merged. In other words, state sharing in $\mathcal{A}$ is not only across programs but also across inputs.

$$[\textsc{Presyn-Var}]\ \frac{x \in \Sigma_{var} \quad s \to x \in R \quad a \in \mathcal{D}_x}{q_s^a \in Q \quad x \in F \quad x \to q_s^a \in \Delta} \qquad [\textsc{Presyn-Final}]\ \frac{q_{s_0}^a \in Q}{q_{s_0}^a \in Q_f}$$

$$[\textsc{Presyn-Prod}]\ \frac{s \to f(s_1, \cdots, s_n) \in R \quad q_{s_i}^{a_i} \in Q \quad a = [\![f]\!]^\sharp(a_1, \cdots, a_n)}{q_s^a \in Q \quad f \in F \quad f(q_{s_1}^{a_1}, \cdots, q_{s_n}^{a_n}) \to q_s^a \in \Delta}$$

Fig. 5. Rules for BuildOfflineFTA, which construct an FTA $\mathcal{A} = (Q, F, Q_f, \Delta)$, given a DSL whose grammar is $G = (V, \Sigma, s_0, R)$ and whose abstract semantics is defined with abstract transformers $[\![\cdot]\!]^\sharp$. $\mathcal{D}_x$ gives the set of abstract values for input variable x (i.e., the space of inputs of interest for x).

Implication on synthesis. Since states in $\mathcal{A}$ are shared across different inputs, simply extracting an *accepting run* (as in prior work [15, 57]) is insufficient. That is, given $e = e_{in} \mapsto e_{out}$, an accepting run ρ that ends in a final state $q_{s_0}^a$, where $e_{out} \in \gamma(a)$, does not guarantee that its corresponding program P satisfies e under abstract semantics. This is because ρ might map an input variable x in P to a state q_s^a, where $a \neq \alpha^{\mathcal{D}_x}(e_{in})$. We will present a solution to this issue in the next Section 3.4.

3.4 Slicing

As alluded to at the end of Section 3.3, given $e = e_{in} \mapsto e_{out}$, we are interested in *specific* accepting runs in $\mathcal{A}$—which not only end in a state consistent with e_{out} but also *begin with states that are consistent with* e_{in}. Such runs are said to be *consistent with* e, which is formally defined below.

Definition 3.1 (Example-consistent run). Given an FTA $\mathcal{A}$ constructed by Figure 5 for a given DSL and an example $e = e_{in} \mapsto e_{out}$, consider an accepting run ρ in $\mathcal{A}$ that corresponds to program P. We say ρ is consistent with e, if the following two conditions are true.

(1) ρ is output-consistent. That is, ρ maps P to a final state $q_{s_0}^a$, such that $e_{out} \in \gamma(a)$.
(2) ρ is input-consistent. That is, ρ maps each input variable x in P to q_s^a, such that $a = \alpha^{\mathcal{D}_x}\big(e_{in}(x)\big)$, i.e., the abstract value a correctly abstracts the concrete value that e_1 binds x to.

It should be obvious that ρ's corresponding program P must satisfy e according to the DSL's abstract semantics that is used to build $\mathcal{A}$.

Oracle-guided slicing algorithm. Now, we are ready to present our Slice algorithm; see Figure 6. Recall how Slice is used in our synthesis algorithm (see line 4 of Algorithm 1): it takes the offline FTA $\mathcal{A}$ and an example e (and an oracle O), and returns a *slice* $\mathcal{A}_e$ which is formally defined below.

Definition 3.2 (Slice). Given an FTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ constructed by Figure 5 for a given DSL and an example e, a slice $\mathcal{A}_e = (Q', F, Q_f', \Delta')$ is a *sub-FTA* of $\mathcal{A}$ (that is, $Q' \subseteq Q$, $Q_f' \subseteq Q_f$, $\Delta' \subseteq \Delta$), such that an accepting run ρ of $\mathcal{A}$ is consistent with e iff ρ is an accepting run of $\mathcal{A}_e$.

Because example-consistent runs always correspond to abstractly consistent programs, $\mathcal{L}(\mathcal{A}_e)$ is exactly the set of programs that satisfy e under the DSL's abstract semantics.

Let us now explain how Figure 6 extracts $\mathcal{A}_e$ from $\mathcal{A}$. Our basic idea is to find states that are involved in at least one example-consistent run of $\mathcal{A}$—such states belong to $\mathcal{A}_e$. The Output rule first identifies final states of $\mathcal{A}_e$: $q_{s_0}^a$ is a final state of $\mathcal{A}_e$, if (i) $q_{s_0}^a$ is a final state in $\mathcal{A}$ where a is consistent with the output example e_{out}; (ii) there is a run in $\mathcal{A}$ to $q_{s_0}^a$ that is consistent with e_{in}. Note that (ii) is checked by InputConsistent, which additionally takes as input the oracle O; we will explain how this check works shortly. The Transition rule serves as the recursive case, which grows $\mathcal{A}_e$ by processing transitions in $\mathcal{A}$. Specifically, given a state q in $\mathcal{A}_e$, we add an incoming transition of q and its argument states $q_1, \cdots, q_n$ to $\mathcal{A}_e$, if InputConsistent holds for every q_i.

$$[\textsc{Output}]\ \frac{q^a_{s_0} \in Q_f \qquad e_{out} \in \gamma(a) \qquad \textsc{InputConsistent}(q^a_{s_0}, e_{in}, O)}{q^a_{s_0} \in Q' \quad q^a_{s_0} \in Q'_f}$$

$$[\textsc{Transition}]\ \frac{q \in Q' \qquad f(q_1, \cdots, q_n) \to q \in \Delta \qquad \forall i \in [1, n] : \textsc{InputConsistent}(q_i, e_{in}, O)}{q_1, \cdots, q_n \in Q' \quad f(q_1, \cdots, q_n) \to q \in \Delta'}$$

$$[\textsc{Input}]\ \frac{q^a_s \in Q' \qquad x \to q^a_s \in \Delta \qquad a = \alpha^{\mathcal{D}_x}\left(e_{in}(x)\right)}{x \to q^a_s \in \Delta'}$$

where $\textsc{InputConsistent}(q, e_{in}, O) = \left(\text{True iff } \alpha(e_{in}) \models O(q)\right)$ for $\alpha(e_{in}) = \left\{x \mapsto \alpha^{\mathcal{D}_x}\left(e_{in}(x)\right)\right\}$

Fig. 6. Rules for Slice, which constructs a slice/FTA $\mathcal{A}_e = (Q', F, Q'_f, \Delta')$, given (i) an FTA $\mathcal{A} = (Q, F, Q_f, \Delta)$, (ii) an input-output example $e = e_{in} \mapsto e_{out}$, and (iii) an oracle O for $\mathcal{A}$.

$$O(q^a_s) = \bigvee_{\delta \in \Delta} \Psi_\delta \quad \textbf{where } \Psi_\delta = \begin{cases} x = a & \text{if } \delta = x \to q^a_s \\ \bigwedge_{i \in [1,n]} O(q_i) & \text{if } \delta = f(q_1, \cdots, q_n) \to q^a_s \end{cases}$$

Fig. 7. Rules for BuildOracle, which constructs an oracle O by processing each state q^a_s in $\mathcal{A}$.

Finally, when reaching an input variable, the Input rule is used to collect those transitions $x \to q^a_s$ where a correctly abstracts x's concrete value in e_{in}.

Now, let us explain how InputConsistent works. Our approach first constructs an oracle O during presynthesis for the offline FTA $\mathcal{A}$ (see line 2 in Algorithm 1). At a high level, O maps every state q in $\mathcal{A}$ to a constraint Ψ: conceptually, Ψ describes, for every run ρ to q, the *abstract input* Γ that ρ is consistent with. Here, Γ is a binding environment that maps input variables to abstract values. In other words, if $O(q)$ does not hold for some Γ, then we know there is no run in $\mathcal{A}$ to q that is consistent with Γ; this is exactly how InputConsistent utilizes O. In particular, as shown in Figure 6, InputConsistent first abstracts the input example e_{in} to an abstract input Γ. Then, the check simply boils down to checking if Γ is a satisfying assignment for $O(q)$.

Building oracle. Figure 7 presents the BuildOracle procedure. For each state q^a_s in $\mathcal{A}$, Ψ_δ creates a constraint for each incoming transition of q^a_s, and their disjunction gives the constraint Ψ for q^a_s. Intuitively, this means, Ψ is true (i.e., there is an input-consistent run to q^a_s), as long as one of its incoming transitions lies on an input-consistent run. Ψ_δ considers two cases. In the base case where $\delta = x \to q^a_s$, Ψ_δ is simply $x = a$, meaning the abstract input must bind x to a in order for an input-consistent run to reach q^a_s via δ. In the recursive case, Ψ_δ conjoins $O(q_i)$ across all argument states. This is because q^a_s is reachable via δ only if all of its argument states are reachable.

Addressing FTA bloat. Let us now take a step back, and revisit the FTA bloat problem. The reason we can significantly reduce the bloat is because our slicing process is guarded by InputConsistent, which ensures all states and transitions added to $\mathcal{A}_e$ are necessary. One straw-man approach is to remove the InputConsistent check: (i) first traverse $\mathcal{A}$ top-down to obtain $\mathcal{A}'$ that contains all output-consistent runs, and (ii) then retain only runs in $\mathcal{A}'$ that are input-consistent. The problem is that (i) will cause bloat, which is exactly what InputConsistent addresses.

Implementation. Our implementation incorporates a couple of optimizations. One is to simplify every constraint Ψ, during BuildOracle (see Figure 7), into its disjunctive normal form, where each disjunct encodes a valid abstract input. This allows Ψ to be represented as a set of abstract

$$[\textsc{Search-Var}]\ \frac{x \to q_s^a \in \Delta \qquad c = e_{in}(x)}{q_s^c \in Q' \quad x \to q_s^c \in \Delta'} \qquad [\textsc{Search-Final}]\ \frac{q_{s_0}^c \in Q' \qquad c = e_{out}}{q_{s_0}^c \in Q'_f}$$

$$[\textsc{Search-Transition}]\ \frac{f(q_{s_1}^{a_1}, \cdots, q_{s_n}^{a_n}) \to q_s^a \in \Delta \qquad q_{s_i}^{c_i} \in Q' \qquad c_i \in \gamma(a_i) \qquad c = [\![f]\!](c_1, \cdots, c_n)}{q_s^c \in Q' \quad f(q_{s_1}^{c_1}, \cdots, q_{s_n}^{c_n}) \to q_s^c \in \Delta'}$$

Fig. 8. Rules for constructing a (concrete) FTA $\mathcal{A}' = (Q', F, Q'_f, \Delta')$, given a slice/FTA $\mathcal{A}_e = (Q, F, Q_f, \Delta)$ for an input-output example $e = e_{in} \mapsto e_{out}$.

inputs; then InputConsistent (see Figure 6) can be done by checking "if $O(q)$ contains $\alpha(e_{in})$". We further optimize the Output rule in Figure 6, to avoid going over every state in Q_f. The idea is to build a map M *offline* for $\mathcal{A}$: M maps each valid abstract input Γ to a set S of final states, such that: for any Γ, $M(\Gamma)$ contains q iff $O(q)$ contains Γ. Then, Output first does a look-up $M\big(\alpha(e_{in})\big)$ to find input-consistent final states, followed by checking against e_{out}.

3.5 Theorem

Theorem 3.3 (Soundness and Completeness). *Suppose $\mathcal{A}_{e_1}$ is the slice returned by Slice at line 4 of Algorithm 1, given a DSL with abstract semantics and a set $E = \{e_1, \cdots, e_n\}$ of examples. Then $\mathcal{L}(\mathcal{A}_{e_1})$ is exactly the set of DSL programs that satisfy e_1 under abstract semantics. That is, Slice is both sound (i.e., every accepting run of $\mathcal{A}_{e_1}$ represents a DSL program that satisfies e_1 under abstract semantics) and complete (i.e., every DSL program that satisfies e_1 under abstract semantics corresponds to an accepting run in $\mathcal{A}_{e_1}$).*

Proof. Proofs can be found in the extended version of this work [13]. □

4 Implementation and Instantiations

This section presents an implementation of our presynthesis-based synthesis framework, followed by instantiations of the framework for three different domains.

4.1 Foresighter: Implementing Presynthesis-Based Synthesis

While the presynthesis module is parametrized with a domain-specific abstraction, our implementation of the synthesis component is domain-general. In particular, our Search algorithm is based on the standard bottom-up search technique with equivalence class reduction [57], which we briefly explain below. Our implementation of the entire framework is called Foresighter.

Recall our Search procedure (see line 5 of Algorithm 1): it takes as input a set E of examples and a slice/FTA $\mathcal{A}_e$ for one example $e \in E$, and returns a program that satisfies all examples in E under concrete semantics. Our implementation constructs an FTA $\mathcal{A}'$ given $\mathcal{A}_e$ and e, using rules from Figure 8. This is a bottom-up process which "concretizes" $\mathcal{A}_e$ using the DSL's concrete semantics: Search-Var builds transitions for x, Search-Transition iteratively grows $\mathcal{A}'$, and finally Search-Final marks final states. Whenever a final state q is produced, we pause the concretization process, and immediately enumerate accepting runs (i.e., queries) to q: if a program passing E is found, then Search terminates; otherwise, concretization resumes, and the aforementioned process repeats. We return *null* if no satisfying program is found (within the given timeout).

4.2 ForesighterSQL: Instantiating Foresighter for SQL

We first show our query DSL, then present the abstraction (i.e., abstract domain and transformers) used for presynthesis, and finally explain key optimizations in brief for the SQL domain.

$$
\begin{array}{rrcl}
\textit{Query} & Q & ::= & x \mid \text{Project}_{L_c}(Q) \mid \text{Filter}_{\phi}(Q) \mid \text{GroupBy}_{L_c,L_g}(Q) \\
 & & \mid & \text{Distinct}_{L_c}(Q) \mid \text{IJoin}_{\phi}(Q,Q) \mid \text{LJoin}_{\phi}(Q,Q) \\
\textit{Column List} & L_c & ::= & [col, \cdots, col] \\
\textit{Aggregation List} & L_g & ::= & [g, \cdots, g] \\
\textit{Aggregation} & g & ::= & agg(col) \\
\textit{Condition} & \phi & ::= & col \odot col \mid col \odot v \mid \phi \wedge \phi \mid \phi \vee \phi \\
\textit{Operator} & \odot & ::= & \leq \mid < \mid = \mid \neq \mid > \mid \geq \mid \text{like}
\end{array}
$$

$x \in$ **Input Variables** $\quad col \in$ **Column Indices** $\quad v \in$ **Values** $\quad agg \in \{\text{Min}, \text{Max}, \text{Sum}, \text{Count}, \text{Avg}\}$

Fig. 9. Syntax of our query DSL (which is an expressive subset of SQL).

Query language. Figure 9 gives the syntax (which is an expressive subset of SQL). A query Q takes as input one or more table-typed variables, and returns a table. Operators all have standard semantics. $\text{Project}_{L_c}(Q)$ selects columns L_c from Q's result, where L_c is a list of column indices. $\text{Filter}_{\phi}(Q)$ retains rows that satisfy a filter condition ϕ. Here, ϕ supports comparison between two columns, or between a column and a value (string or numeric). $\text{GroupBy}_{L_c,L_g}(Q)$ first groups all rows from Q by L_c, then performs aggregation for each group based on L_g, and finally outputs a table with the aggregated content for all groups. $\text{Distinct}_{L_c}(Q)$ removes duplicate rows according to L_c. $\text{IJoin}_{\phi}(Q_1, Q_2)$ and $\text{LJoin}_{\phi}(Q_1, Q_2)$ perform inner join and left join respectively. When ϕ is used as a join condition, without loss of generality, we assume the column index col that appears on the left (resp. right) hand side of $\odot$ always refers to a column in Q_1 (resp. Q_2); this is solely to simplify the later presentation of our abstract semantics.

Abstract domain. Like prior work [56], we apply predicate abstraction, which abstracts a concrete value into a *conjunction* of predicates. A predicate always refers to a grammar symbol s whose value is being abstracted, and may also involve constants and input variables. Specifically, we define the following predicate *templates* to abstract a query Q's return table.

- Our first predicate template is $\texttt{NumRows}^{Q}(\sigma_1, \sigma_2)$, where Q is the *Query* grammar symbol and σ_i is the summation over some input variables and constants. This predicate is true, if the number of rows in Q falls in the range $[\sigma_1, \sigma_2]$.
- Our second template is $\texttt{NumCols}^{Q}(k)$, where k is a constant. It means "query Q yields a table with exactly k columns".
- The next one is $\texttt{SubsetOf}^{Q}(J, x)$, where J is a set of column indices (constants) and x is an input variable. This predicate is true, if $(\forall j \in J)$ the j-th column of Q's return table is a subset of some column from x's table (i.e., the table that x binds to).
- The last template is $\texttt{ColType}^{Q}(J, t)$, stating that $(\forall j \in J)$ the j-th column of Q has type t. Here, t is either numeric or string.

Abstracting L_g and L_c boils down to abstracting the column index col. This is done by a predicate $\texttt{In}^{col}(J)$, meaning col is in the set J of column indices. Then, L_c is abstracted by $\texttt{In}^{L_c}([J_1, \cdots, J_l])$, meaning L_c has exactly l columns and its i-th column $(\forall i \in [1, l])$ is in J_i. L_g can be abstracted in a similar manner. These abstractions are crucial for propagating Q's abstract values through query operators (which heavily make use of L_c and L_g).

The join/filter condition ϕ is abstracted using $\texttt{MaxCols}^{\phi}(m_1, m_2)$, where m_1 (resp. m_2) is the max column index involved in ϕ that appear on the left (resp. right) hand side of $\odot$. Specifically, any col used in ϕ is still abstracted by $\texttt{In}$. Then, m_1 (resp. m_2) for ϕ is computed by taking the max of $\texttt{In}$ predicates for each col that is on the left (resp. right) hand side of any $\odot$ in ϕ.

Finally, ForesighterSQL's abstract input space (namely, $\mathcal{D}_x$ in Figure 5) considers up to 3 input variables each with up to 15 columns. In other words, k in $\texttt{NumCols}$ ranges from 1 to 15.

$$[\![\text{Project}]\!]^\sharp(\texttt{NumCols}^{\mathcal{Q}}(k),\ \texttt{In}^{L_c}([J_1,\cdots,J_l])) := \texttt{NumCols}^{\mathcal{Q}'}(l)\ \text{ if } \textstyle\bigwedge_{i\in[1,l]}(\max(J_i)\le k)$$
$$[\![\text{Project}]\!]^\sharp(\texttt{SubsetOf}^{\mathcal{Q}}(J,x),\ \texttt{In}^{L_c}([J_1,\cdots,J_l])) := \texttt{SubsetOf}^{\mathcal{Q}'}(J',x)\ \text{ where } J'=\{i\mid J_i\subseteq J\}$$
$$[\![\text{Project}]\!]^\sharp(\texttt{ColType}^{\mathcal{Q}}(J,t),\ \texttt{In}^{L_c}([J_1,\cdots,J_l])) := \texttt{ColType}^{\mathcal{Q}'}(J',t)\ \text{ where } J'=\{i\mid J_i\subseteq J\}$$

$$[\![\text{Filter}]\!]^\sharp(\texttt{NumRows}^{\mathcal{Q}}(\sigma_1,\sigma_2),\ \phi) := \texttt{NumRows}^{\mathcal{Q}'}(0,\sigma_2)$$
$$[\![\text{Filter}]\!]^\sharp(\texttt{NumCols}^{\mathcal{Q}}(k),\ \texttt{MaxCols}^{\phi}(m_1,m_2)) := \texttt{NumCols}^{\mathcal{Q}'}(k)\ \text{ if } \max(m_1,m_2)\le k$$
$$[\![\text{Filter}]\!]^\sharp(\texttt{ColType}^{\mathcal{Q}}(J,t),\ \phi) := \texttt{ColType}^{\mathcal{Q}'}(J,t)$$

$$[\![\text{IJoin}]\!]^\sharp(\texttt{NumRows}^{\mathcal{Q}_1}(\sigma_{1,1},\sigma_{1,2}),\ \texttt{NumRows}^{\mathcal{Q}_2}(\sigma_{2,1},\sigma_{2,2}),\ \phi) := \texttt{NumRows}^{\mathcal{Q}'}(0,\sigma_{1,2}\times\sigma_{2,2})$$
$$[\![\text{IJoin}]\!]^\sharp(\texttt{NumCols}^{\mathcal{Q}_1}(k_1),\ \texttt{NumCols}^{\mathcal{Q}_2}(k_2),\ \texttt{MaxCols}^{\phi}(m_1,m_2)) := \texttt{NumCols}^{\mathcal{Q}'}(k_1+k_2)\ \text{ if } m_i\le k_i$$
$$[\![\text{IJoin}]\!]^\sharp(\texttt{ColType}^{\mathcal{Q}_1}(J,t),\ \mathcal{Q}_2,\ \phi) := \texttt{ColType}^{\mathcal{Q}'}(J,t)$$

$$[\![\texttt{In}^{col_1}(J_1)\odot\texttt{In}^{col_2}(J_2)]\!]^\sharp := \texttt{MaxCols}^{\phi}(\max(J_1),\max(J_2))$$
$$[\![\texttt{MaxCols}^{\phi_1}(m_{1,1},m_{1,2})\wedge\texttt{MaxCols}^{\phi_2}(m_{2,1},m_{2,2})]\!]^\sharp := \texttt{MaxCols}^{\phi}(\max(m_{1,1},m_{2,1}),\max(m_{1,2},m_{2,2}))$$

Fig. 10. Representative abstract transformers for operators in Figure 9.

Abstract transformers. Following prior work [56], we use a generic transformer for each operator f, which takes an abstract value—that is a *conjunction* of predicates—for each of f's arguments and returns an abstract value. Therefore, it suffices to define an abstract transformer for every operator f given an *atomic* predicate for each argument. Figure 10 gives a representative list of transformers.

$$[\![f]\!]^\sharp\Big(\bigwedge_{i_1}\varphi_{i_1},\cdots,\bigwedge_{i_n}\varphi_{i_n}\Big) := \bigwedge_{i_1}\cdots\bigwedge_{i_n}[\![f]\!]^\sharp(\varphi_{i_1},\cdots,\varphi_{i_n})$$

Optimizations. ForesighterSQL incorporates a few optimizations to improve the performance for both presynthesis and synthesis. First, we reduce the abstract domain by defining a canonical order on column indices in `SubsetOf` and `ColType` (e.g., numeric columns before string columns). This does not hinder completeness, since the desired order can always be recovered with a projection, but it significantly speeds up presynthesis. On top of this canonicalization, we slightly adjust our Search to find a query Q whose return table *subsumes* all columns in the example output table, and then infer the top-level projection given Q. Our final optimization is around further accelerating presynthesis. Recall that a set J of columns is tracked in our abstraction. The canonical column order allows us to represent J as an interval, making presynthesis faster without losing completeness.

4.3 ForesighterString: Instantiating Foresighter for String Transformation

While SQL is the focus domain, to fully demonstrate our generality, we also briefly present two more instantiations, one for string transformation and another for matrix manipulation (next section). We use DSLs from the Blaze work [56], and focus our presentation on the abstraction.

Same as Blaze, we use a predicate template to track the length of strings. Our second template $\texttt{FromInp}^S(J,x)$ tracks if the j-th character ($\forall j\in J$) of string S appears in the input variable x. The third and last template $\texttt{CharEq}^S(J,x,k)$ refines $\texttt{FromInp}^S(J,x)$: it requires the j-th character in S to match the k-th character in x. The atomic transformers can be found in the extended version [13].

4.4 ForesighterMatrix: Instantiating Foresighter for Matrix Manipulation

Following Blaze, we have a predicate template $\texttt{DimNum}^T(k)$ to track the number of dimensions for a matrix/tensor, and another $\texttt{Shape}^T(v)$ that tracks the tensor's shape (i.e., the i-th dimension of tensor T is $v[i]$). We also have a third template $\texttt{ElemEq}^T(v_{out},x,v_{in})$, indicating the element in T at position v_{out} is equal to the element in an input tensor x at position v_{in}. The corresponding abstract transformers can be found in the extended version [13].

5 Evaluation

This section presents a series of experiments designed to answer the following questions. Note that the first four questions are centered around our focus domain (i.e., SQL), while RQ5 is designed to evaluate the generality of Foresighter (in string and matrix domains).

- **RQ1**: How effective is ForesighterSQL at solving example-based SQL synthesis tasks?
- **RQ2**: How does ForesighterSQL compare with state-of-the-art techniques?
- **RQ3**: How important are the key ideas (namely, presynthesis and oracle)?
- **RQ4**: How does ForesighterSQL scale as the underlying abstract semantics becomes finer?
- **RQ5**: How does ForesighterString and ForesighterMatrix perform in practice?

SQL Benchmarks. We collected all 3,866 benchmarks from Cubes [1, 6, 7], which, to the best of our knowledge, contains all example-based SQL synthesis benchmarks in the literature [37, 48, 51, 53] to date. Each benchmark consists of a "ground-truth" query P and one input-output example. We augmented each benchmark with 16 additional examples, following the same setup in Cubes [7]. This is because the original example led to many synthesized queries that are unintended (i.e., not equivalent to P). At the end, we curated 3,817 benchmarks[3], each with 17 examples in total.

String and matrix benchmarks. We use the same benchmarks from prior work [56]. In particular, we have 108 SyGuS string transformation benchmarks and 39 matrix manipulation benchmarks.

Testbed. Our experiments are conducted on machines with Xeon Gold 6154 CPU and 192GB RAM running Red Hat 8.10. SQLite 3.46 is used to evaluate SQL queries.

5.1 RQ1: ForesighterSQL Results

Setup. To evaluate ForesighterSQL, we first run its presynthesizer to build an offline FTA $\mathcal{A}$ and the oracle O. Then, given a benchmark with a set E of 17 examples, we run ForesighterSQL's synthesizer (given $\mathcal{A}$, O, and E), and record: (i) whether or not the benchmark is solved (i.e., if a query that satisfies E is found) within our timeout (10 minutes), and (ii) if so, the synthesis time. During this process, we also log detailed information (such as time breakdown and FTA size).

End-to-end results. Let us first go over ForesighterSQL's end-to-end performance; see Table 1a. Overall, it successfully solved 2,604 benchmarks using a median of 0.01 seconds. 82% of them can be solved within 5 seconds. The average solving time is 22.0 seconds.

Synthesis results. Table 1b and Table 1c further present the synthesis time breakdown. While Slice always terminates in 0.5 seconds (see Table 1b), Search time tends to be longer (see Table 1c). Despite this, Search can still finish in 5 seconds for 82% of the benchmarks.

Let us dive into more details, beginning with Table 1c. Recall that Search (i) first builds a concrete FTA (CFTA), based on concrete semantics, for the given slice, and (ii) then enumerates accepting runs of this CFTA and along the way checking their corresponding queries against all examples, until a satisfying program is found. It turns out that, for most of the benchmarks, over 60% of the total Search time ends up being spent on SQLite-related work. The resulting CFTA is typically small, with a median of 160 states; and for over 72% of the benchmarks, it has under 10K states. Not all such states, however, lead to a final state. Only those accepting runs (i.e., reaching a final state) are enumerated and checked. For most benchmarks, the first accepting run yields a query that passes all 17 examples; and for over 88% of them, no more than 1,000 queries are checked.

Now let us turn to Table 1b. The main take-away is that our slices are small, with a median of 37 states and 47 transitions. For over 90% of the benchmarks, the slice has under 1,000 states and 3,000

[3] 49 benchmarks were not included, since their ground-truth queries were not executable on at least one of the inputs.

Table 1. ForesighterSQL results. First, the end-to-end benchmark-solving results are summarized in Table 1a. Then, Slice and Search (i.e., two main components of synthesis) results are given in Table 1b and Table 1c, respectively. Finally, Table 1d reports presynthesis results (for BuildOfflineFTA and BuildOracle).

(a) End-to-end results. Time statistics are calculated over all solved benchmarks.

#Solved	Time (sec)	
	median	*avg*
2,604	0.01	22.0

(b) Slice results across all solved benchmarks. We first present the (median, average, max) slicing time. Then, we give the (median, average, max) number of states in the slice, followed by the same statistics for transitions.

Total Slice time (sec)			#States			#Transitions		
median	*avg*	*max*	*median*	*avg*	*max*	*median*	*avg*	*max*
0.001	0.01	0.5	37	234	5.0K	47	1.6K	93.2K

(c) Search results across all solved benchmarks. We first present the (median, average, max) total search time across solved benchmarks. We then report the (median, average, max) percentage of the total time spent on SQLite-related work (e.g., evaluating queries and communicating with SQLite). Next, "#States in CFTA" presents the (median, average, max) number of states, upon termination of Search, that are created while concretizing the slice into a concrete FTA (CFTA). Finally, "#Queries checked" reports the (median, average, max) number of accepting runs of the CFTA that are enumerated, each corresponding to a query.

Total Search time (sec)			SQLite time / total time			#States in CFTA			#Queries checked		
median	*avg*	*max*	*median*	*avg*	*max*	*median*	*avg*	*max*	*median*	*avg*	*max*
0.01	22.0	581.2	64.7%	60.7%	96.6%	160	108.2K	8.6M	1	32.8K	6.9M

(d) Presynthesis results. For BuildOfflineFTA, we report the FTA construction time and the resulting FTA's size. For BuildOracle, we report the oracle construction time and its size.

BuildOfflineFTA				BuildOracle	
Time	Size on disk	#States	#Transitions	Time	Size on disk
5.0 hrs	51.0 GB	454.8K	192.6M	7.2 hrs	16.8 GB

transitions. In addition to the small size, slicing does not require running abstract transformers—we just collect states/transitions from the offline FTA. This also explains why Slice is very fast.

Presynthesis results. As shown in Table 1d, our offline FTA can be built in 5.0 hours, consuming 51.0 GB of disk space. Its oracle takes 7.2 hours to be constructed and occupies 16.8GB of disk space. We believe this resource requirement is affordable.

Failure analysis. Despite significantly stronger performance than state-of-the-art, there are 1,213 benchmarks (out of 3,817 total) that ForesighterSQL was not able to solve (within 10 minutes). A closer inspection reveals that DSL expressiveness is a major reason, accounting for 653 of the unsolved benchmarks. Notable unsupported features include HAVING clause (149 benchmarks), scalar subqueries (98), INTERSECT (97), EXCEPT (66), and complex LIMIT (60). Another common cause is FTA concretization timeout (543 benchmarks). Ground-truth queries for them typically have complicated structures (such as many levels of JOINs and complex filter predicates), and some of them involve long SQLite execution time (due to large input/intermediate tables). Other reasons include non-determinism in certain operators (such as ORDER BY LIMIT, which, in the presence of ties, may return a different output table from the user-provided one), and abstract input space coverage (e.g., input example has more columns than the max our offline supports).

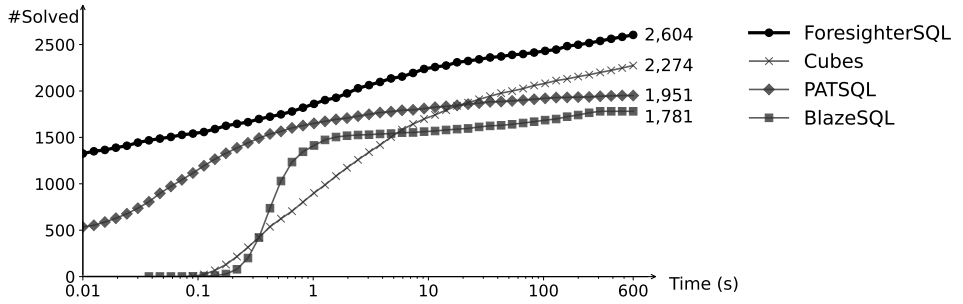


Fig. 11. Number of solved benchmarks when varying the timeout (up to 600 seconds), for ForesighterSQL and all baselines. A data point (X, Y) means Y benchmarks can be solved using X seconds as the timeout. Note that y-axis is in linear scale while we use log scale for x-axis. We also label the number of benchmarks solved (with 600 seconds as the timeout) for each tool, right next to its line.

5.2 RQ2: ForesighterSQL vs. State-of-the-Art

Baselines. We compare ForesighterSQL with *all* state-of-the-art *bespoke* SQL synthesizers (to the best of our knowledge). The first one is PatSQL [48], which combines program sketching and enumeration. The other is Cubes [1, 6, 7], which uses SMT-based enumeration with SQL-specific pruning and search hints (such as aggregation functions to be used; which we do not require).

Besides SQL synthesizers, we also include a third baseline, called BlazeSQL, which implements the abstraction-refinement-based synthesis framework from the Blaze paper [56]. In particular, BlazeSQL first builds an FTA $\mathcal{A}_0$ using an initial (coarse) abstraction. $\mathcal{A}_i$ is then refined into $\mathcal{A}_{i+1}$, by incorporating new predicates mined from a spurious program extracted from $\mathcal{A}_i$. This iterative refinement process terminates when a program that passes all 17 examples is extracted. BlazeSQL uses the same DSL as ForesighterSQL, as well as the same predicate templates and transformers.

Setup. We use the same setup (as in RQ1) to run all baselines (using the same 10-minute timeout). For each baseline, we record the solved benchmarks and their corresponding solving times.

Results. Figure 11 summarizes the comparison. ForesighterSQL consistently solves more benchmarks than all baselines. Using a 600s timeout, it solves 15% more than the best baseline, Cubes. This improvement is significant for a few reasons. First, ForesighterSQL is an instantiation of our framework with a pretty standard (online) search technique, whereas Cubes is specialized to SQL and incorporates domain-specific guidance. Second, Cubes uses additional hints (e.g., candidate aggregation functions), which ForesighterSQL does not use. Finally, for those benchmarks solved by both, ForesighterSQL is significantly faster, with a median of 90× speed-up and a geometric mean of 51×. On the other hand, Cubes can solve 235 benchmarks that ForesighterSQL cannot. 87 of them involve unsupported features (e.g., certain set operations). For the rest, ForesighterSQL timed out during concretization. This suggests clear room to further improve our approach.

The next best baseline is PatSQL. ForesighterSQL significantly outperforms it by solving 33% more benchmarks. These results highlight the advantage of abstraction-based pruning and our presynthesis-based technique over purely online synthesizers. While BlazeSQL utilizes abstract semantics, it actually solves the least number of benchmarks. In particular, it spends up to 391K refinement steps to solve a benchmark, with an average of 8K steps across solved benchmarks. This may suggest that, in our case, predicates discovered locally do not prune away many programs.

Table 2. ForesighterSQL vs. ablations. We first present the number of solved benchmarks. Then, we report the (median, average) synthesis time across all solved benchmarks. The next column compares ForesighterSQL with each ablation over benchmarks solved by both. Finally, we present the Slice time for each tool.

	#Solved	Total time (sec)		Median speed-up over ablation	Slice time (sec)		
		median	*avg*		*median*	*avg*	*max*
ForesighterSQL	2,604	0.01	22.0	–	0.001	0.01	0.5
NoPresyn	2,587	9.3	40.1	413.1×	9.3	21.5	246.8
NoOracle	2,539	34.1	85.0	614.4×	34.0	73.6	405.5

5.3 RQ3: Ablation Studies

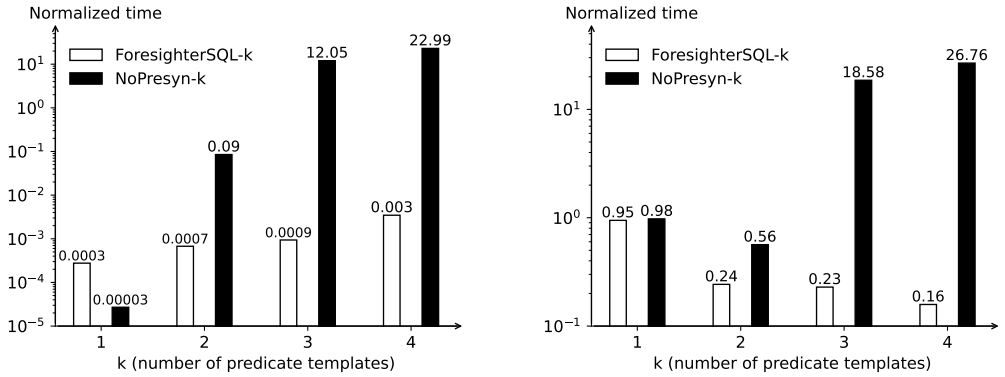
Ablations. Our ablations aim to evaluate the two underpinning ideas of our framework—namely, presynthesis and oracle-based slicing. The first ablation, called NoPresyn, removes the presynthesis step: that is, it performs Slice, given e_1 and the grammar, by building $\mathcal{A}_{e_1}$ *from scratch*. In other words, NoPresyn is a standard FTA-based synthesizer using abstract semantics (see Section 2.1). Our second ablation, NoOracle, still has presynthesis but does not build the oracle. Instead, the slice $\mathcal{A}_{e_1}$ is computed by traversing the offline FTA using two passes. The first bottom-up pass obtains all states reachable from e_1's input, followed by a second top-down pass which removes unnecessary states (that do not reach an output-consistent final state).

Setup. We use the same setup as in RQ2 to run all ablations.

ForesighterSQL vs. NoPresyn. See results in Table 2. Let us first compare ForesighterSQL with NoPresyn. Overall, ForesighterSQL solves more benchmarks in less time. This is primarily due to NoPresyn's significant slowdown on Slice. Building the slice online during synthesis can take up to 247 seconds. As illustrated in Section 2.2, this slowdown is caused by the FTA bloat issue; that is, an excessive number of unnecessary states and transitions are created (online). A more detailed analysis of this bloat will be presented in the next Section 5.4.

ForesighterSQL vs. NoOracle. We observe a similar comparison with NoOracle, which is also due to Slice being slow. Careful readers might wonder why it is even slower than NoPresyn, given that NoOracle has additional access to an offline FTA $\mathcal{A}$ (while NoPresyn builds the slice *from scratch*). The reason is that the offline $\mathcal{A}$ includes states for all inputs (not just e_1's input). This makes $\mathcal{A}$ large, to the extent that computing $\mathcal{A}_{e_1}$ by *traversing* $\mathcal{A}$ (without our oracle) ends up being more expensive than building $\mathcal{A}_{e_1}$ from scratch! Specifically, recall that NoOracle uses a bottom-up pass to collect states q in $\mathcal{A}$ that are reachable from e_1's input. During this pass, it iteratively processes q's outgoing transitions to discover new reachable states. But the vast majority of such transitions are "illegal": that is, at least one argument state is not reachable from e_1's input. Processing such transitions ends up wasting much of the time. So our take-away is: only building an offline $\mathcal{A}$ is far from sufficient; we must also use $\mathcal{A}$ wisely (our oracle seems to be a good way).

Sensitivity to abstract input space. Recall that ForesighterSQL considers up to 3 input tables each with up to 15 columns. To understand the sensitivity of ForesighterSQL's performance to this tuning knob, we create two ablations, one by shrinking the space down to up to 10 columns and the other with up to 20. The latter is able to express only a handful of additional benchmarks, but the corresponding offline FTA is 1.3× bigger than ForesighterSQL and oracle 2.1× bigger. The former, on the other hand, solves 2,478 benchmarks (95% of those solved by ForesighterSQL), whose offline FTA is 3.4× smaller and oracle 7.1× smaller. Further reducing the abstract input space (e.g., up to 5 columns) would significantly hinder the solving capability.



(a) Normalized Slice time (y-axis) vs. k (x-axis).

(b) Normalized total time (y-axis) vs. k (x-axis).

Fig. 12. Online scaling for ForesighterSQL-k and NoPresyn-k, with k from 1 to 4. Figure 12a shows the scaling for Slice, and Figure 12b presents the scaling for the end-to-end synthesis process (i.e., Slice + Search). Both figures consider benchmarks solved by ForesighterSQL-4 (which solves the most). For an unsolved benchmark, we use the timeout (600s) as the "solving time" to plot Figure 12b, and use the corresponding slicing time to plot Figure 12a. For each benchmark, all tools' times (both slicing and total) are normalized to NoPresyn-0's *total* time. Finally, for both figures, we present the geometric mean of normalized times across all benchmarks. For instance, ForesighterSQL-4's normalized total time is 0.16 in Figure 12b, which means its total time is 16% of NoPresyn-0's (representing a geometric mean of 6.3× speed-up). Finally, $k = 0$ is not presented: their slicing time should be interpreted as 0 and total time as 1, after normalization.

5.4 RQ4: Scaling

Our hypothesis: "scaling law" for synthesis. Akin to the "scaling law" [27] in deep learning—that is, model performance seems to keep improving as model size, dataset size, and computational resources spent on training increase—we hypothesize that a similar phenomenon also exists in the context of (search-based) program synthesis. Specifically, our hypothesis is that, in principle, finer-grained abstraction should lead to faster synthesis, *provided sufficient computational resources*. We believe that presynthesis is a promising building block towards realizing this "scaling law" for synthesis. Hence, this section will present empirical data on how synthesis scales as a function of abstraction granularity, aiming to provide preliminary validation for our hypothesis.

Setup. We create 5 versions of ForesighterSQL with *increasing* abstraction granularity. Specifically, we consider the following ordered list of predicate templates:

`NumRows`, `NumCols`, `SubsetOf`, `ColType`.

ForesighterSQL-k (k ranging from 0 to 4) is a variant of ForesighterSQL, that uses the first k templates to build the offline FTA. For instance, ForesighterSQL-0 essentially performs Search on the given grammar, without any abstraction-based pruning. ForesighterSQL-2 tracks a table's "shape" (i.e., number of rows and columns) as the abstraction, but not table content. ForesighterSQL-4 (i.e., ForesighterSQL) uses the finest abstraction, which additionally tracks if a column is a subset of some input table and the column's data type.

As a comparison, we also create 5 versions of NoPresyn in the same fashion as above, where NoPresyn-k uses the same abstraction as ForesighterSQL-k. This allows us to see how a purely online FTA-based approach scales, and compare it with the presynthesis-based counterpart.

Table 3. Peak slice size change, when increasing k from 1 to 4, for ForesighterSQL-k and NoPresyn-k. The "Peak #states" column reports that, for each tool, the number of states *ever* created by Slice. Same as Figure 12, we consider all benchmarks solved by ForesighterSQL-4. Given a benchmark, each tool's peak number of states is normalized to ForesighterSQL-0's number. Finally, for each tool, we report the geometric mean across all benchmarks. The "Peak #transitions" column is calculated in the same manner but for transitions.

	Peak #states				Peak #transitions			
	$k=1$	$k=2$	$k=3$	$k=4$	$k=1$	$k=2$	$k=3$	$k=4$
ForesighterSQL-k	1.15	1.06	1.13	1.37	1.11	0.98	1.06	1.50
NoPresyn-k	2.16	16.83	468.42	1533.07	2.31	17.24	404.89	1445.50

Table 4. Offline scaling for ForesighterSQL.

(a) Offline artifact scaling when increasing k from 1 to 4.

		$k=1$	$k=2$	$k=3$	$k=4$
Offline FTA	#States	1.5K	21.2K	62.0K	454.8K
	#Transitions	36.7K	1.1M	2.4M	192.6M
	Size on disk	4MB	238MB	504MB	51.0GB
	Build time	0.2s	7s	27s	5.0h
Offline oracle	Size on disk	5MB	13MB	4.7GB	16.8GB
	Build time	1s	33s	71s	7.2h

(b) State split factor statistics from k to $k+1$, across all states in offline FTA $\mathcal{A}_k$ with k templates. For instance, increasing k from 1 to 2, a state in $\mathcal{A}_1$ is split into a median of 17 states in $\mathcal{A}_2$.

	$1 \to 2$	$2 \to 3$	$3 \to 4$
median	17	2	72
75th	45	4	882
90th	79	6	3,375

Note that ForesighterSQL-0 and NoPresyn-0 are the same—because they both directly perform Search over the grammar (without presynthesis). They also serve as a very good "baseline", which performs standard bottom-up search (using FTA-based equivalence-class reduction) [57]. Thus, we will normalize other tools' results to theirs, to facilitate easy inspection of the scaling trend.

Finally, for each ForesighterSQL-k, we first run its presynthesizer and then use its synthesizer to solve benchmarks (using the same 10-minute timeout). The same timeout is used for NoPresyn-k.

Slice time and total time vs. abstraction granularity. Figure 12a presents how Slice scales as the abstract semantics becomes more granular, and Figure 12b shows how the total time scales. As shown by Figure 12a, NoPresyn's Slice time skyrockets as k increases, whereas ForesighterSQL is much more scalable with consistently low slicing time. This also explains why ForesighterSQL's total time continues to improve (see Figure 12b) as the abstraction granularity increases, whereas NoPresyn's total time has a U-shaped scaling curve.

Peak slice size vs. abstraction granularity. Table 3 presents how the peak slice size changes as the abstract semantics is more granular. As we can see, ForesighterSQL's slice size remains quite stable, whereas NoPresyn's explodes with increasing k. This is exactly the FTA bloat phenomenon that Section 2.2 illustrates: (i) while NoPresyn-k eventually gives the same slice as ForesighterSQL-k, it creates many unnecessary states and transitions, causing the bloat; (ii) this bloat is exacerbated, when finer-grained abstract semantics is used. This is exactly why NoPresyn's slicing time scales poorly in Figure 12a, eventually leading to the U-shaped scaling curve of total time in Figure 12b.

Offline scaling. Switching gears, Table 4 presents ForesighterSQL's offline scaling. Comparing adjacent columns in Table 4a, one might notice the offline FTA grows unevenly. For example, the number of transitions from 1 to 2 increases roughly by 30×, but from 2 to 3 only about 2×, and

Table 5. ForesighterString-k and ForesighterMatrix-k results, in comparison with Blaze and NoPresyn.

		Online		Offline					
		Synthesize		BuildOfflineFTA				BuildOracle	
		#Solved	Avg. time (s)	Time (s)	Size (disk)	#States	#Transitions	Time (s)	Size (disk)
String	$k=1$	82	0.751	0.01	1.0 MB	764	17.8K	0.03	212.6 KB
	$k=2$	90	0.107	4.3	338.0 MB	1.6M	3.3M	6.9	427.4 MB
	$k=3$	90	0.094	504.0	8.0 GB	35.4M	75.1M	116.6	6.5 GB
	Blaze	90	0.296	–	–	–	–	–	–
	NoPresyn	90	3.883	–	–	–	–	–	–
Matrix	$k=1$	39	12.376	0.002	52.8 KB	118	685	0.0006	6.92 KB
	$k=2$	39	0.101	66.6	3.8 GB	307.7K	67.2M	177.7	206.8 MB
	$k=3$	39	0.086	472.1	20.4 GB	4.1M	354.7M	424.0	1.4 GB
	Blaze	39	1.449	–	–	–	–	–	–
	NoPresyn	39	6.698	–	–	–	–	–	–

then from 3 to 4 around 80×. This is because from 2 to 3, the abstraction precision increment is not as much as the other two, with 3 to 4 being the most. To better quantify the granularity change, instead of using only k, we measure the "state split factor" from k to $k+1$. That is, for each state q in $\mathcal{A}_k$ (offline FTA built with k templates), how many states q' in $\mathcal{A}_{k+1}$ that q get split to (i.e., the abstract value in q' refines that in q). Table 4b reports some statistics. As we can see, the split factor grows the most from 3 to 4, which explains the uneven FTA growth observed in Table 4a.

5.5 RQ5: ForesighterString and ForesighterMatrix Results

Table 5 reports main results for ForesighterString and ForesighterMatrix.

Comparing with Blaze. For both domains, our technique (using the finest abstraction; i.e., $k=3$) solves the same set of benchmarks as Blaze [56] and is faster.

Importance of presynthesis. Presynthesis is crucial, as NoPresyn is significantly slower.

Online and offline scaling. A similar online scaling trend as in SQL domain is observed: increasing k leads to faster online synthesis. On the other hand, the offline cost also grows, which is consistent with the offline scaling behavior for SQL. Overall, for both domains, the offline overhead is affordable even with our finest (i.e., $k=3$) abstraction.

6 Related Work

Abstraction-based synthesis. Abstract semantics has been widely used in the program synthesis literature, across a broad range of domains, such as data transformation [15, 17, 56], SQL query generation [53], syntax-guided synthesis [21, 60], among others [14, 20, 35]. Building upon the literature, we contribute a new method that models a DSL's abstract semantics in an offline phase.

Representation-based synthesis. Our abstract semantics modeling is carried out by constructing an FTA $\mathcal{A}$, such that every accepting run of $\mathcal{A}$ reflects a program's execution on a potential abstract input. This is related to a long line of work on representation-based synthesis, including the seminal FlashFill work [18] (which uses version space algebras as the underlying representation) among many others [16, 20, 22, 28, 30, 34, 36, 56–58] (which utilize other representations, such as simple

graphs, hypergraphs, Petri nets, and FTAs). Our novelty lies in *when* to build the representation. We build it offline for semantics modeling, whereas virtually all prior work builds it online.

Graph reachability in program synthesis. Our FTAs can be viewed as a form of hypergraphs, and our synthesis algorithm can be viewed as solving a form of graph search problem. Hence, our approach is also related to prior work that uses a graph-based formulation of synthesis. Notable examples include Prospector [34], Blaze [56], SyPet [16], Hoogle+ [20], among others [26, 43]. Besides technical differences (e.g., the specific problem formulation), a major distinction lies in the scale of the graph. For instance, unlike Prospector which builds a "small" graph (8 MB on disk), our FTA is significantly larger (51 GB on disk). This motivates new graph search techniques, as opposed to reusing standard ones (as in Prospector).

Reachability oracle. Given this large offline-built FTA, our online search algorithm leverages a form of reachability oracle [9, 49] (a precomputed data structure to help efficiently answer online queries about if a path exists between any two nodes in a graph). Such oracles have been used for efficiently processing queries over massive databases (e.g., in biomedical and social networks) [8, 24, 31–33, 39, 44, 52]. To the best of our knowledge, we are the first to develop reachability oracles for FTAs and in the context of program synthesis.

Preprocessing. At a high level, our work leverages preprocessing (or precomputation) to accelerate online synthesis. This is related to a large body of work that applies preprocessing in different areas (e.g., quantum computing [23, 42], cryptography [2, 3], and algorithms [38, 50]). Preprocessing has also been used in programming languages—e.g., to help efficiently answer queries about context-free language reachability [31–33] in the context of program analysis. In addition to precomputing our oracle, building the offline FTA can also be viewed as preprocessing (i.e., transforming a given DSL into a large FTA). To our best knowledge, we are the first to use preprocessing for (search-based) synthesis. While *neural* synthesis techniques (e.g., based on LLMs) can also be viewed as having a preprocessing component (i.e., training the neural model), they are quite different from our work: we model the DSL's abstract semantics, whereas they model the distribution of programs.

7 Conclusion and Discussion

We developed the first *presynthesis-based* synthesis framework, and implemented it in Foresighter. The evaluation on three domains demonstrated promising results. We hope that our work could fuel the next paradigm shift, from crafting sophisticated online synthesis algorithms to using compute resources (e.g., offline time/space, online memory/parallelism), for further synthesis scale-up.

Necessity of presynthesis. Thoughtful readers might wonder if our offline presynthesis phase is necessary: for instance, would a purely online approach that *incrementally* grows a "global" FTA (shared across tasks) *on-the-fly*, for every incoming synthesis task, be fast enough? We suspect that a trivial incremental approach would be much slower (as it still suffers from the FTA bloat issue). A better solution might exist, but we believe it is non-trivial and its performance is also unclear. For example, using an *overly engineered* abstract input space that covers *exactly* our SQL benchmarks (i.e., "overfitting"), the resulting FTA takes 70 minutes to build. This time would be spread across all benchmarks, causing a significant slowdown for many of them. Note that this time is the theoretical lower bound, *assuming no additional overhead*—which seems quite hard to realize. For instance, one challenge is, given a new task, how to maximally avoid visiting existing transitions and creating additional ones. A further problem is how to incrementally update the oracle on-the-fly. We believe these challenges could motivate new incremental algorithms for synthesis (potentially built upon our presynthesis idea), which is an interesting future direction to pursue.

Acknowledgments

We would like to thank the anonymous PLDI reviewers, for their insightful feedback. We thank Greg Bodwin and Nicole Wein, for helping us navigate the literature on graph reachability oracle and providing useful pointers. We would also like to extend our thanks to Chenglong Wang and Yuepeng Wang, for their comments on an earlier draft of this work; and to Anhong Guo, Nicole Wein, and Xu Wang, for their help with experimentation resources. This research was supported by the National Science Foundation (NSF) under Grant Numbers CCF-2210832, CCF-2318937, CCF-2236233, and CCF-2123654. Zheng Guo was supported by Schmidt Sciences.

Data-Availability Statement

The artifact that implements the techniques and supports the evaluation results reported in this paper is available on Zenodo [12].

References

[1] CUBES tool. https://github.com/OutSystems/CUBES.
[2] Elliptic curve point multiplication. https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication.
[3] Rainbow table (wiki). https://en.wikipedia.org/wiki/Rainbow_table.
[4] Syntax-Guided Synthesis website. https://sygus-org.github.io/.
[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. doi:10.1109/FMCAD.2013.6679385
[6] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco Manquinho, and Ruben Martins. 2024. Towards reliable SQL synthesis: Fuzzing-based evaluation and disambiguation. In *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 232–254.
[7] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco Manquinho, and Ruben Martins. 2025. CUBES: A Parallel Synthesizer for SQL Using Examples. *Form. Asp. Comput.* (Sept. 2025). doi:10.1145/3768578
[8] Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu Yu. 2022. DLCR: efficient indexing for label-constrained reachability queries on large dynamic graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1645–1657.
[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
[10] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree automata techniques and applications.
[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Los Angeles, California, 238–252. doi:10.1145/512950.512973
[12] Rui Dong, Qingyue Wu, Danny Ding, Zheng Guo, Ruyi Ji, and Xinyu Wang. 2026. *Foresighter (Presynthesis: Towards Scaling Up Program Synthesis with Finer-Grained Abstract Semantics)*. doi:10.5281/zenodo.19078478
[13] Rui Dong, Qingyue Wu, Danny Ding, Zheng Guo, Ruyi Ji, and Xinyu Wang. 2026. Presynthesis: Towards Scaling Up Program Synthesis with Finer-Grained Abstract Semantics. arXiv:TBD
[14] Azadeh Farzan and Victor Nicolet. 2021. Counterexample-guided partial bounding for recursive function synthesis. In *International Conference on Computer Aided Verification*. Springer, 832–855.
[15] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.
[16] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 599–612.
[17] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. doi:10.1145/2737924.2737977
[18] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. doi:10.1145/1926385.1926423
[19] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
[20] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[21] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. 2023. Absynthe: Abstract interpretation-guided synthesis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1584–1607.
[22] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 27–38.
[23] William J Huggins and Jarrod R McClean. 2023. Accelerating Quantum Algorithms with Precomputation. *arXiv preprint arXiv:2305.09638* (2023).
[24] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 595–608.
[25] Keith JC Johnson, Rahul Krishnan, Thomas Reps, and Loris D'Antoni. 2024. Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 935–961.
[26] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314.
[27] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).
[28] James Koppel, Zheng Guo, Edsko De Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching entangled program spaces. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 23–51. doi:10.1145/3547622
[29] 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. doi:10.1145/3434335
[30] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient bottom-up synthesis for programs with local variables. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1540–1568.
[31] Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient algorithms for dynamic bidirected dyck-reachability. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.
[32] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.
[33] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2022. Fast graph simplification for interleaved-dyck reachability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–28.
[34] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005), 48–61.
[35] Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2024. Optimal Program Synthesis via Abstract Interpretation. *Proc. ACM Program. Lang.* 8, POPL, Article 16 (Jan. 2024), 25 pages. doi:10.1145/3632858
[36] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29. doi:10.1145/3498682
[37] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES: A SQL synthesizer using query reverse engineering. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2853–2856.
[38] Mihai Patrascu and Liam Roditty. 2010. Distance oracles beyond the Thorup-Zwick bound. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 815–823.
[39] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825.
[40] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.
[41] 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*. 107–126.
[42] Gokul Subramanian Ravi, Pranav Gokhale, Yi Ding, William Kirby, Kaitlin Smith, Jonathan M Baker, Peter J Love, Henry Hoffmann, Kenneth R Brown, and Frederic T Chong. 2022. CAFQA: A classical simulation bootstrap for variational quantum algorithms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 15–29.
[43] Alex Reinking and Ruzica Piskac. 2015. A type-directed approach to program repair. In *International Conference on Computer Aided Verification*. Springer, 511–517.
[44] Sherif Sakr and Ghazi Al-Naymat. 2012. An overview of graph indexing and querying techniques. *Graph Data Management: Techniques and Applications* (2012), 71–88.
[45] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.
[46] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 313–326.
[48] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. 2021. PATSQL: efficient synthesis of SQL queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.* 14, 11 (July 2021), 1937–1949. doi:10.14778/3476249.3476253
[49] Mikkel Thorup. 2004. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM (JACM)* 51, 6 (2004), 993–1024.
[50] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *Journal of the ACM (JACM)* 52, 1 (2005), 1–24.
[51] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 535–548.
[52] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 345–358.
[53] 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*. 452–466.
[54] Xinyu Wang. 2019. *An efficient programming-by-example framework*. Ph. D. Dissertation.
[55] Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L McMillan. 2018. Learning Abstractions for Program Synthesis. In *International Conference on Computer Aided Verification*. Springer, 407–426.
[56] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program Synthesis using Abstraction Refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. doi:10.1145/3158151
[57] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26. doi:10.1145/3133886
[58] Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational program synthesis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27. doi:10.1145/3276525
[59] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *VLDB* 11, 5 (2018), 580–593. doi:10.1145/3187009.3177735
[60] 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 2025-11-13; accepted 2026-04-03