Mobius: Synthesizing Relational Queries with Recursive and Invented Predicates

AALOK THAKKAR, University of Pennsylvania, USA
NATHANIEL SANDS, University of Southern California, USA
GEORGE PETROU, University of Southern California, USA
RAJEEV ALUR, University of Pennsylvania, USA
MAYUR NAIK, University of Pennsylvania, USA
MUKUND RAGHOTHAMAN, University of Southern California, USA

Synthesizing relational queries from data is challenging in the presence of recursion and invented predicates. We propose a fully automated approach to synthesize such queries. Our approach comprises of two steps: it first synthesizes a non-recursive query consistent with the given data, and then identifies recursion schemes in it and thereby generalizes to arbitrary data. This generalization is achieved by an iterative predicate unification procedure which exploits the notion of data provenance to accelerate convergence. In each iteration of the procedure, a constraint solver proposes a candidate query, and a query evaluator checks if the proposed program is consistent with the given data. The data provenance for a failed query allows us to construct additional constraints for the constraint solver and refine the search. We have implemented our approach in a tool named Möbius. On a suite of 21 challenging recursive query synthesis tasks, Möbius outperforms three state-of-the-art baselines GenSynth, ILASP, and Popper, both in terms of runtime and accuracy. We also demonstrate that the synthesized queries generalize well to unseen data.

CCS Concepts: • **Information systems** → Relational database query languages; • **Software and its engineering** → Automatic programming; Programming by example; • **Theory of computation** → Constraint and logic programming.

Additional Key Words and Phrases: Programming by example, example-guided synthesis, recursive program synthesis

ACM Reference Format:
Aalok Thakkar, Nathaniel Sands, George Petrou, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2023. Mobius: Synthesizing Relational Queries with Recursive and Invented Predicates. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 271 (October 2023), 25 pages. https://doi.org/10.1145/3622847

1 INTRODUCTION

The synthesis of relational queries from input-output examples is a challenging and foundational problem in program synthesis. Ideally, we would like a technique that is simultaneously: (*a*) scalable enough to be applicable to real-world instances, (*b*) expressive in terms of the kinds of queries that it can synthesize, and (*c*) fully automatic, so it requires minimal guidance from non-expert users. Significant progress has been made on this problem in recent years [Cropper et al. 2020], and a variety of algorithms have been proposed, including algorithms based on evolutionary

Authors' addresses: Aalok Thakkar, athakkar@seas.upenn.edu, University of Pennsylvania, , USA; Nathaniel Sands, njsands@usc.edu, University of Southern California, , USA; George Petrou, gpetrou@usc.edu, University of Southern California, , USA; Rajeev Alur, alur@cis.upenn.edu, University of Pennsylvania, , USA; Mayur Naik, mhnaik@cis.upenn.edu, University of Pennsylvania, , USA; Mukund Raghothaman, raghotha@usc.edu, University of Southern California, , USA.

2475-1421/2023/10-ART271
https://doi.org/10.1145/3622847

search [Mendelson et al. 2021], numerical relaxation [Si et al. 2019], constraint solving [Cropper and Morel 2021; Law et al. 2020], and counterexample-guided search [Raghothaman et al. 2020].

Despite these strides, one particularly difficult class of queries consists of those that simultaneously require recursion and *invented predicates*, i.e., intermediate concepts which are not explicitly provided as part of the training data. Such queries are useful in many application domains, such as bioinformatics, knowledge discovery, program analysis, and software-defined networking. Complex queries in these domains often involve a recursive core followed by additional qualifying rules. For example, contact tracing in an epidemic might involve following the chain of exposures and subsequently accounting for the subset of asymptomatic and vulnerable populations.

To further illustrate the challenge with synthesizing such queries, consider the task of learning the concept of strongly connected components (SCC) in directed graphs. In this case, the user would provide examples of graphs via their adjacency relations, $\mathtt{edge}(x, y)$, and identify pairs of vertices in the same SCC. We may express the target relation scc using the following relational query, $Q_{\mathtt{scc}}$, in Datalog syntax:

$$\begin{aligned}
\mathtt{scc}(x, y) &\text{ :- } \mathtt{path}(x, y), \mathtt{path}(y, x). \\
\mathtt{path}(x, z) &\text{ :- } \mathtt{path}(x, y), \mathtt{path}(y, z). \\
\mathtt{path}(x, y) &\text{ :- } \mathtt{edge}(x, y).
\end{aligned}$$

Here, the relation path corresponds to the transitive closure of edge, and scc is defined as the conjunction of path and its inverse. The query is recursive and the tuples of the invented relation path are not specified by the user either in the input or output data. Although several existing tools, including GenSynth [Mendelson et al. 2021], ILASP [Law et al. 2020], and Popper [Cropper and Morel 2021] can synthesize $Q_{\mathtt{scc}}$, they typically require additional instance-specific guidance in the form of candidate rules, signatures of the invented predicates, or bounds on the size of the search space. In this paper, we study the fully automatic synthesis of such queries.

We begin by observing that existing techniques such as the enumerative search-based tool Scythe [Wang et al. 2017c] and example-guided synthesis (EGS) [Thakkar et al. 2021] can synthesize arbitrarily complex *non-recursive* queries without any instance-specific guidance from the user. Furthermore, although these non-recursive query synthesis algorithms are successful in inferring patterns in data of finite size, they have limited ability to generalize these patterns to perform computation on arbitrarily large datasets. In contrast, techniques to synthesize recursive queries are successful in generalizing patterns once the user constrains the space of candidate programs. Our key insight is to combine the strengths of both paradigms in a synthesis tool that is scalable, targets an expressive fragment of queries, and offers end-to-end automation.

Our approach is a two-phase synthesis engine called Möbius which we show in Figure 1. In the first phase, we synthesize a non-recursive query Q using a template-free technique, example-guided synthesis (EGS) [Thakkar et al. 2021]. In the second phase, we use this non-recursive query to constrain the hypothesis space to queries that *generalize* it. This procedure uses provenance-guided reasoning to *unify* invented predicates. Since these invented predicates may not directly manifest in Q itself, we propose a normalization procedure that exposes them by rewriting Q into a semantically equivalent query $\overline{Q}$. Then, generalization proceeds in an iterative fashion that involves synergistic interaction between a constraint solver and a query evaluator. In each iteration, the former selects a candidate unification μ, and the latter checks whether the resulting query $\mu(\overline{Q})$ is consistent with the given input-output data. If so, the process terminates; otherwise, the constraints are updated to avoid the ill-fated unification choice and the process is repeated.

A naive constraint formulation suffers from prohibitively slow convergence in practice due to an exponential number of unification choices. To accelerate convergence, a standard technique is to

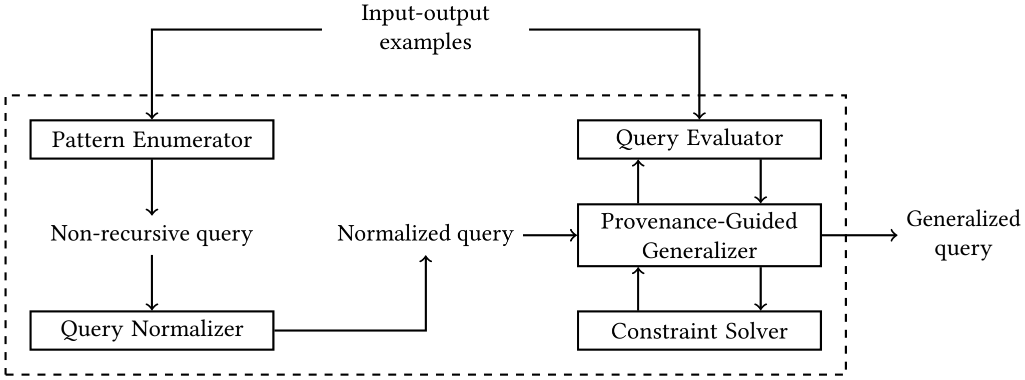


Fig. 1. The architecture of the Möbius synthesis engine. We start by using a pattern enumerator (such as EGS) to generate a non-recursive query that is consistent with the input-output examples, and then generalize it into a recursive query using a provenance-guided generalization algorithm. This procedure repeatedly uses a constraint solver to generate candidate solutions whose consistency it determines using a query evaluator. Analysis of failed candidate solutions result in additional constraints that are fed back to the constraint solver thereby pruning the search space in subsequent iterations.

use conflict-driven learning which iteratively prunes the search space [Feng et al. 2018a]. Motivated by it, in the second phase, we develop a novel provenance-guided technique that leverages *data provenance* [Cheney et al. 2009; Zhao et al. 2020]—a derivation tree that serves as a witness of a given spurious tuple—to identify a minimal incorrect core of the ill-fated unification choice. We thereby eliminate from future consideration all other unification choices that are similarly destined to derive the spurious tuple.

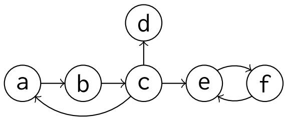
We evaluate Möbius on 21 tasks from the literature with a diverse range of recursion schemes, including queries with transitive closure, and linear, non-linear, and mutually recursive predicates. The tool successfully discharges all problem instances within a total of 128 seconds. We also compare with three state-of-the-art tools GenSynth, ILASP, and Popper that employ evolutionary search and constraint solving techniques. Given sufficient parallelization, GenSynth can solve all benchmarks in 796 seconds, while ILASP and Popper solve only 13 and 8 tasks, in 1057 and 59 seconds respectively.

In summary, our work makes the following contributions:

(1) We propose a fully automated approach to synthesize relational queries with both recursion and invented predicates.
(2) We propose a novel unification procedure to identify recursion schemes in non-recursive programs, thereby generalizing them beyond the training data. The procedure efficiently identifies unification constraints by leveraging data provenance in relational queries.
(3) We have implemented Möbius, an end-to-end synthesis tool and demonstrate that it outperforms state-of-the-art approaches on a variety of challenging tasks in terms of synthesis time as well as the generalizability of the synthesized programs.

2 OVERVIEW

We begin with a high-level overview of our end-to-end synthesis framework. As a running example, we consider the task of synthesizing a query that computes the relation induced by the strongly connected components (SCCs) in a directed graph.



(a) Graph G.

Input I	
edge(a,b),	edge(b,c),
edge(c,a),	edge(c,d),
edge(c,e),	edge(e,f),
edge(f,a)	

(b) Input edge relation.

Positive labels O^+:		
scc(a,a),	scc(a,b),	scc(a,c),
scc(c,b),	scc(e,f),	scc(f,e),
Negative labels O^-:		
scc(a,d),	scc(c,d),	scc(c,e),
scc(d,e),	scc(c,f),	scc(e,c)

(c) Positive and negative labels for scc.

Fig. 2. The synthesis task is specified as a search for a relational query P that takes the graph G as an input and returns a set of pairs of vertices O such that O is a superset of O^+ and disjoint from O^-. We call such a query consistent with the input-output examples.

2.1 Problem Setting

Figure 2a shows a directed graph and Figure 2b describes its adjacency relation edge. A user can provide this relation as an input I to a synthesis engine with the intent to synthesize a query that computes a relation scc representing SCCs in the graph. In order to express this intent, they label some pairs of vertices as positive tuples O^+ and some as negative tuples O^- such that the tuples in O^+ must be present in relation scc while those in O^- must be absent. Figure 2c shows an example of the positive and negative labelled output tuples. The synthesis task is to find a query P *consistent* with (I, O^+, O^-), that is, a query that takes I, the edge relation, as an input and generates all tuples in O^+ but none in O^-.

Example 2.1. The following relational query $P^{\leq 3}_{\mathsf{scc}}$ is consistent with (I, O^+, O^-):

$$\begin{aligned} r_1 &: \mathsf{scc}(x,x) \text{ :- } \mathsf{edge}(x,y), \mathsf{edge}(y,x). \\ r_2 &: \mathsf{scc}(x,y) \text{ :- } \mathsf{edge}(x,y), \mathsf{edge}(y,x). \\ r_3 &: \mathsf{scc}(x,x) \text{ :- } \mathsf{edge}(x,y), \mathsf{edge}(y,z), \mathsf{edge}(z,x). \\ r_4 &: \mathsf{scc}(x,y) \text{ :- } \mathsf{edge}(x,y), \mathsf{edge}(y,z), \mathsf{edge}(z,x). \\ r_5 &: \mathsf{scc}(x,z) \text{ :- } \mathsf{edge}(x,y), \mathsf{edge}(y,z), \mathsf{edge}(z,x). \end{aligned} \tag{1}$$

$P^{\leq 3}_{\mathsf{scc}}$ is a collection of rules $\{r_1, \dots, r_5\}$. We can interpret each rule in $P^{\leq 3}_{\mathsf{scc}}$ as a Horn clause. For instance, the second rule means that if both tuples (x, y) and (y, x) are in the edge relation, then vertices x and y are in the same SCC. We formally define the syntax and semantics of relational queries in Section 3.1.

As we can observe, $P^{\leq 3}_{\mathsf{scc}}$ correctly captures all SCCs in the graph of Figure 2a, and in general, in all directed graphs with SCCs of size 2 or 3. A program synthesis technique such as example-guided synthesis (EGS) can efficiently synthesize such queries. However, the goal of the synthesis task is to find a query that is not only consistent with the user input, but also generalizes to match the intent of the user.

2.2 Synthesis of Recursive Queries

We next illustrate a query for computing SCCs that matches user intent.

Example 2.2. The following relational query P_{scc} computes SCCs in a given graph:

$$\begin{aligned} r'_1 &: \mathsf{scc}(x,y) \text{ :- } \mathsf{path}(x,y), \mathsf{path}(y,x). \\ r'_2 &: \mathsf{path}(x,z) \text{ :- } \mathsf{path}(x,y), \mathsf{path}(y,z). \\ r'_3 &: \mathsf{path}(x,y) \text{ :- } \mathsf{edge}(x,y). \end{aligned} \tag{2}$$

Observe that P_{scc} uses a predicate path which is not pre-defined (that is, it does not occur as an input to the synthesis task) and also calls itself in rule r_2'. A predicate that does not appear in the synthesis task as an input or an output predicate is called an *invented* predicate. A predicate that can *call* itself by applying a series of rules is called a *recursive* predicate. Our goal is the discovery of succinct and general queries such as P_{scc} that potentially use invented and recursive predicates.

Further, observe that a non-recursive query synthesis engine such as EGS or Scythe cannot generate P_{scc}, nor can we modify them to directly enumerate such a query as they do not support recursive or invented predicates. Two principal challenges arise when synthesizing such queries: *First*, the outputs of intermediate relations are under-constrained and are not explicitly specified in the input-output examples. This significantly inhibits the ability of the synthesizer to prune candidate queries during search. *Second*, synthesis engines which attempt to enumerate candidate programs also need constraints on the number and schema of these intermediate predicates. Tools such as ProSynth and ILASP that support recursion would require additional supervision in form of the correct set of mode declarations that specify the invented and recursive predicates with their schema. Although GenSynth is able to discover invented predicates, it implicitly assumes that they must share schema with one of the input or output predicates already provided as part of the problem description. In our experiments in Section 6, we will present benchmarks that require both *predicate invention* as well as *schema invention*, and observe that state-of-the-art tools fail to correctly synthesize these queries.

We leverage a non-recursive query $P^{\leq 3}_{\mathsf{scc}}$ that can be generated without templates (by using EGS) as a starting point for the search for P_{scc}. Observe that P_{scc} *generalizes* $P^{\leq 3}_{\mathsf{scc}}$. That is, on any graph P_{scc} will also report all pairs of vertices generated by $P^{\leq 3}_{\mathsf{scc}}$. In addition, for graphs with SCCs of size 4 or more, P_{scc} can report pairs of vertices $\mathsf{scc}(x, y)$ that $P^{\leq 3}_{\mathsf{scc}}$ would miss.

In order to generalize it, we first *normalize* the given query, that is, convert it into a semantically equivalent query where a premise comprises of at most one input predicate or two invented predicates. We describe this process in Section 3.2. For ease of notation, let $Q = P^{\leq 3}_{\mathsf{scc}}$. The normal form $\overline{Q}$ for the query Q would look like:

$$\begin{aligned}
\rho_1 &: \mathsf{scc}(x, x) \text{ :- } R_1(x, y), R_1(y, x). \\
\rho_2 &: \mathsf{scc}(x, y) \text{ :- } R_1(x, y), R_1(y, x). \\
\rho_3 &: \mathsf{scc}(x, x) \text{ :- } R_1(x, y), R_2(y, x). \\
\rho_4 &: \mathsf{scc}(x, y) \text{ :- } R_1(x, y), R_2(y, x). \\
\rho_5 &: \mathsf{scc}(x, y) \text{ :- } R_2(x, y), R_1(y, x). \\
\rho_6 &: R_2(x, z) \text{ :- } R_1(x, y), R_1(y, z). \\
\rho_7 &: R_1(x, y) \text{ :- } \mathsf{edge}(x, y).
\end{aligned} \tag{3}$$

Observe that rules $\rho_1, \ldots, \rho_5$ in $\overline{Q}$ correspond exactly to the rules $r_1, \ldots, r_5$ in Q, and uses two invented predicates R_1 and R_2. The rules for these invented predicates are ρ_6 and ρ_7.

At this point, we can highlight our key insight. There is a correspondence between the rules ρ_6 and ρ_7 in $\overline{Q}$ and the rules r_2 and r_3 in P_{scc} by which the rules are identical up to renaming of the predicates. That is, if we could map $R_1(x, y)$ and $R_2(x, y)$ to $\mathsf{path}(x, y)$, we would obtain the rules r_2 and r_3 in P_{scc}. Furthermore, applying this mapping to each of the rules of $\overline{Q}$ would give us a query similar to P_{scc}.

Our generalization technique builds on this insight and identifies an efficient way to search for maps that *unify* invented predicates. One happy effect of the normalization process is that it automatically discovers the schema of the intermediate relations, thus eliminating the need for them to be explicitly provided as a part of the input. In this sense, the normalized program $\overline{Q}$

effectively serves as a template and constraints the space of candidate programs to those that can be generated by unification.

2.3 Provenance-Guided Generalization

In order to search for a query P that generalizes $\overline{Q}$, we seek ways to unify the invented predicates R_1 and R_2. Section 5 details a way to encode this as a constraint satisfaction problem. We start with a bound on the number of invented predicates. For the sake of this example, let the bound be $k = 1$. That is, we wish to map both R_1 and R_2 to the same predicate, say S_1. Clearly, there are four ways to permute the variables for R_1 and R_2, and each of them gives us a map:

$$\begin{aligned}
\mu_1 &: R_1(x,y) \mapsto S_1(x,y), \quad R_2(x,y) \mapsto S_1(x,y) \\
\mu_2 &: R_1(x,y) \mapsto S_1(x,y), \quad R_2(x,y) \mapsto S_1(y,x) \\
\mu_3 &: R_1(x,y) \mapsto S_1(y,x), \quad R_2(x,y) \mapsto S_1(x,y) \\
\mu_4 &: R_1(x,y) \mapsto S_1(y,x), \quad R_2(x,y) \mapsto S_1(y,x)
\end{aligned}$$

In order to apply a unification μ to $\overline{Q}$, we replace each occurrence of $R_1(x,y)$ and $R_2(x,y)$ with $\mu(R_1(x,y))$ and $\mu(R_2(x,y))$ respectively in each rule. For example, on applying μ_2 to $\overline{Q}$ we get the query $T(\overline{Q},\mu_2)$:

$$\begin{aligned}
\mu_2(\rho_1) &: \texttt{scc}(x,x) \text{ :- } S_1(x,y), S_1(y,x). \\
\mu_2(\rho_2) &: \texttt{scc}(x,y) \text{ :- } S_1(x,y), S_1(y,x). \\
\mu_2(\rho_3) &: \texttt{scc}(x,x) \text{ :- } S_1(x,y), S_1(x,y). \\
\mu_2(\rho_4) &: \texttt{scc}(x,y) \text{ :- } S_1(x,y), S_1(x,y). \\
\mu_2(\rho_5) &: \texttt{scc}(x,y) \text{ :- } S_1(y,x), S_1(y,x). \\
\mu_2(\rho_6) &: S_1(x,z) \text{ :- } S_1(y,x), S_1(z,y). \\
\mu_2(\rho_7) &: S_1(y,x) \text{ :- } \texttt{edge}(x,y).
\end{aligned}$$

Observe that if a tuple is produced by $\overline{Q}$, then for any unification map μ, the same tuple can be generated by $T(\overline{Q},\mu)$ by applying the corresponding set of rules. We formally prove this in Theorem 5.3. However, it is possible for $T(\overline{Q},\mu)$ to have an output larger than $\overline{Q}$. In this sense, unification leads to generalization. We call $T(\overline{Q},\mu)$ a candidate query.

We then check if the candidate query $T(\overline{Q},\mu)$ is consistent with the input-output example (I, O^+, O^-). If it is, then we can return it as a synthesized result. On the other hand, it is also possible that such a generalization is *too general*, that is, it also generates some of the tuples in O^-.

In the example above, the tuple scc(c, d) can be generated by $T(\overline{Q},\mu_2)$ while c and d are not in the same SCC. By analyzing the derivation tree for the erroneously generated tuple, scc(c, d), we can assign *blame* to a part of the unification map. This blame can be converted into a constraint that rules out future unification maps where similar patterns would occur. Blame analysis thereby guides the search towards correct generalizations.

In order to implement this, we set up an interactive process involving a constraint solver that proposes candidate queries and a query evaluator that verifies whether the candidate is consistent with the input-output example. In the case where the candidate is not consistent, the query evaluator provides a derivation tree of every tuple in $t \in O^-$ that can be generated by the candidate. We use these derivation trees to craft the constraints. We further discuss derivation trees in Section 3.1 and the provenance-guided technique in Section 5.3.

The key insight of the provenance-guided technique is to leverage the derivation tree of an unexpected tuple. This tree allows us to precisely identify properties of the unification which led to the unexpected tuple being derived. This allows us to avoid considering such unifications in future iterations.

Eventually, the constraint solver proposes the unification map μ_1. Not only is the query $T(\overline{Q}, \mu_1)$ consistent with (I, O^+, O^-), it is also *similar* to the intended query P_{scc}. It has the rules:

$$\begin{aligned}
\mu_1(\rho_1) &: \mathsf{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).\\
\mu_1(\rho_2) &: \mathsf{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).\\
\mu_1(\rho_3) &: \mathsf{scc}(x, x) \text{ :- } S_1(x, y), S_1(y, x).\\
\mu_1(\rho_4) &: \mathsf{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).\\
\mu_1(\rho_5) &: \mathsf{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).\\
\mu_1(\rho_6) &: S_1(x, z) \text{ :- } S_1(x, y), S_1(y, z).\\
\mu_1(\rho_7) &: S_1(x, y) \text{ :- } \mathsf{edge}(x, y).
\end{aligned}$$

The rules $\mu_1(\rho_2)$, $\mu_1(\rho_5)$, and $\mu_1(\rho_6)$ correspond exactly to the rules r'_1, r'_2, and r'_3 in P_{scc} (Equation 2). The rules $\mu_1(\rho_4)$, and $\mu_1(\rho_5)$ are identical to $\mu_1(\rho_1)$ or can be derived using it. Using the rules $\mu_1(\rho_1)$ and $\mu_1(\rho_5)$, one can *derive* the rule $\mu_1(\rho_1)$ and $\mu_1(\rho_3)$. Once simplified, this gives a correct and interpretable solution to the problem originally posed in Figure 2.

3 PROBLEM FORMULATION

In this section, we overview the syntax and semantics of relational queries, define the Relational Query Synthesis Problem, and discuss its decidability and complexity.

3.1 Syntax and Semantics of Relational Queries

As discussed in the overview, a relational query Q is a set of rules. To define the syntax of rules, we first fix a set of *input* predicates, a set of invented predicates, and a set of output predicates. Each predicate R is associated with an *arity* k. A *literal*, $R(v_1, v_2, \dots, v_k)$, consists of a k-ary predicate R with a list of k variables.

Then, a rule r is of the form:

$$R_h(\vec{u}_h) \text{ :- } R_1(\vec{u}_1), R_2(\vec{u}_2), \dots, R_n(\vec{u}_n),$$

where the single literal on the left, $R_h(\vec{u}_h)$, is the *head* of r and $R_1(\vec{u}_1)$, $R_2(\vec{u}_2)$, …, $R_n(\vec{u}_n)$, is called the *body* of r. The literals in the body can have input predicates, invented predicates, or output predicates, while the head of the rules must have either invented predicates or output predicates. A variable that occurs in the head must appear at least once in the body in order for the variable to be bound. The number of literals in the body of a rule is called the *size* of the rule.

The semantics of a relational query may be specified in multiple equivalent ways; see [Abiteboul et al. 1995] for an overview. In this paper, we will formalize their semantics using rule instantiations and derivation trees. We first fix a data domain D, whose elements we will call *constants*. For example, the set of constants for the input in Figure 2 is the set of vertices of G, that is $D = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}\}$. A *tuple*, $R(c_1, c_2, \dots, c_k)$, consists of a k-ary relation name R with a list of constants, $c_1, \dots, c_k$.

Given a map v from variables to the data domain D, we can *instantiate* a rule by consistently replacing its variables x with constants $v(x)$:

$$R_h(v(\vec{u}_h)) \impliedby R_1(v(\vec{u}_1)), R_2(v(\vec{u}_2)), \dots, R_n(v(\vec{u}_n)).$$

Given a query P and a valuation of the input relations I, a *derivation tree* of a tuple t is a labelled rooted tree where: (*a*) each node of the tree is labeled by a tuple, (*b*) each leaf is labeled by a tuple in I; (*c*) the root node is labeled by t; and (*d*) for each internal node labeled α, there exists an instantiation $\alpha \impliedby \beta_1, \dots, \beta_n$ of a rule in P such that the children of the node are respectively

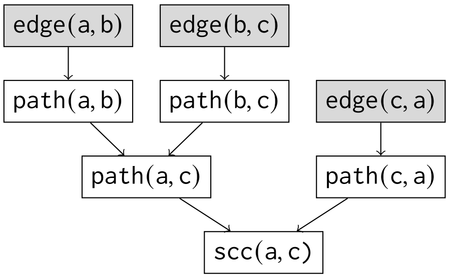


Fig. 3. Example derivation tree of the output tuple scc(a, c) for the query P_{scc}. The input to the query is the graph of Figure 2a.

Query semantics ($[\![P_{\text{scc}}]\!](I)$):

scc(a,a),	scc(a,b),	scc(a,c),
scc(b,a),	scc(b,b),	scc(b,c),
scc(c,a),	scc(c,b),	scc(c,c),
scc(e,e),	scc(e,f),	scc(f,e)
scc(f,f)		

Fig. 4. Semantics of P_{scc} with respect to the input of directed graph G as in Figure 2a. The set I is the set of input tuples and the query semantics are $[\![P_{\text{scc}}]\!](I)$.

labelled $\beta_1, \ldots, \beta_n$. We say that a query P derives t using I if there exists a derivation tree for t. Figure 3 shows the derivation tree for scc(a, b) in P_{scc}. The output $[\![P]\!](I)$ of a query P given an input I is the set of output tuples $R(c_1, c_2, \ldots, c_k)$ which it derives from I. The query P_{scc} on the input in Figure 2a generates the output as in Figure 4.

3.2 Minimal Generalization Problem

Our ultimate goal is to synthesize a recursive relational query which is consistent with given input-output examples. Given a set of input tuples, I, and a set of output tuples partitioned as O^+ and O^-, tools such as EGS and Scythe can effectively synthesize queries P such that P generates tuples in O^+ and does not generate any tuple in O^-. However, these are non-recursive queries. As discussed in the overview, we are interested in the generalization problem where given a query Q that is consistent with the input-output examples, we wish to find a query P that generalizes it. For this purpose, we first define subsumption:

Definition 3.1 (Subsumption). A relational query P subsumes a relational query Q if for any set of input tuples I, $[\![Q]\!](I) \subseteq [\![P]\!](I)$.

That is, for any input I, if Q generates a tuple t, then P also generates t. For example, the query P_{scc} subsumes $P^{\leq 3}_{\text{scc}}$. We also define the size of a relational query as the sum of the sizes of its rules. For example, the size of the query P_{scc} is 5 and of $P^{\leq 3}_{\text{scc}}$ is 11. We can now state the minimal generalization problem:

Problem 3.2 (Minimal Generalization)**.** *Given an input-output example $E = (I, O^+, O^-)$, and a relational query Q consistent with E, find a relational query P that subsumes Q, is consistent with E, and is of minimal size among such queries.*

For example, the user may specify the input and output tuples as described in Section 2, and seek a query that *explains* the relation between input and output tuples. They may use the query $P^{\leq 3}_{\text{scc}}$ generated by EGS as a seed in order to search for the query P_{scc} that uses invented and recursive predicates so it can match the user intent.

4 THE SYNTHESIS ALGORITHM

In this section we describe the end-to-end Möbius algorithm, which takes an input-output example $E = (I, O^+, O^-)$ as input and returns a relational query P (which potentially has invented and recursive predicates). Algorithm 1 summarises the procedure.

Algorithm 1 Möbius(I, O^+, O^-), where (I, O^+, O^-) is an instance of the synthesis task.

```
(1) Let $Q_0 = \text{EGS}(I, O^+, O^-)$. If EGS fails to return a relational query, end the procedure and
    return unsat.
(2) Initialize $Q := \emptyset$.
(3) While there is a tuple $t \in O^+ \setminus [\![Q]\!](I)$:
    (a) Let $r \in Q_0$ derive $t$. Update $Q := Q \cup \{r\}$.
    (b) Let $\overline{Q} = \text{Normalize}(Q)$.
    (c) Compute $P = \text{Generalize}(\overline{Q}, I, O^-)$.
    (d) If $O^+ \subseteq [\![P]\!](I)$, end the procedure and output $t$.
```

We start with using a non-recursive query synthesizer EGS. The output of EGS, Q_0, is a non-recursive query. We construct a query $Q \subseteq Q_0$ on demand, initialized to the empty set, that grows till the synthesized query is not consistent with (I, O^+, O^-).

In order to generalize the query Q, we first normalize it to $\overline{Q}$. We discuss the normalization procedure in Section 4.2. The normalized query is then provided as an input to the provenance-guided generalization procedure which we discuss in Section 5.

4.1 Example-guided Synthesis

Example-guided Synthesis (EGS) is a template-free algorithm to synthesize non-recursive queries from input-output examples. While EGS supports features such as multi-way joins and unions, it does not allow for invented or recursive predicates. Therefore, on inherently recursive tasks, EGS cannot synthesize the *intended* query. $P^{\leq 3}_{\text{scc}}$ is an example of a query that EGS may generate. Additionally, EGS cannot be modified to generate recursive programs as it not a syntax-guided tool.

However, EGS does provide a completeness guarantee that if there exists a non-recursive query consistent with the input-output example (I, O^+, O^-), then EGS will find a consistent query Q_0. We can prove that there exists a recursive relational query consistent with a given input-output example $E = (I, O^+, O^-)$ only if there is a non-recursive query that is consistent with E. Therefore, when EGS returns unsat, we can conclude that there does not exist a query (recursive or non-recursive) that is consistent with the input-output example. Using this, we can prove:

Theorem 4.1 (Completeness). *If there exists a query consistent with the input-output example $E = (I, O^+, O^-)$, then Möbius produces a relational query P consistent with E.*

This is because we use EGS as a first step in the process, which allows Möbius to ensure that if there is no consistent query, then we do not proceed with a futile search. On the other hand, if EGS produces a query, Möbius only further generalizes it and in the worst case, it may output the same query (after normalization). This allows us to conclude a completeness guarantee for the end-to-end synthesis procedure.

4.2 Normalization

Once we have the query Q_0, we construct Q on demand. Then, in Step 3b, we normalize Q to $\overline{Q}$. Normalization introduces invented predicates in the query which we further use for generalization through unification. The following definition of a normal query is motivated by the Chomsky Normal Form for context-free languages [Sipser 2012].

Definition 4.2 (Normal Form). A relational query is said to be in the normal form if every rule is of one of the two forms:

$$R(\vec{x}) \quad \text{:-}\ R_1(\vec{x}_1), R_2(\vec{x}_2)$$

$$R(\vec{x}) \quad \text{:-}\ R_{in}(\vec{x}_{in})$$

where R, R_1, and R_2 are invented predicates and R_{in} is an input predicate. That is, the body of a rule either has two invented predicates or one input predicate.

For example, $P^{\leq 3}_{\mathsf{scc}}$ in Equation 1 is not in normal form while P_{scc} in Equation 2 is. Analogous to context-free languages, the normalization of relational queries can be carried out by rewriting the rules into semantically equivalent rules and introducing invented predicates, and we can show that every query can be *normalized*. We employ a greedy heuristic to normalize queries that allows us to minimize the size of the number of invented predicates as well as their arity.

Let a given rule be of the form:

$$R(\vec{x}) \text{ :- } R_1(\vec{x}_1), R_2(\vec{x}_2), \dots, R_n(\vec{x}_n).$$

We partition the literals in the body into two disjoint sets S_l and S_r such that the number of variables shared by literals in S_l and literals in S_r are minimal. Let $\vec{x}_l$ be a vector of variables that occur in the literals in S_l and either in $\vec{x}$ or a literal in S_r. Similarly, let $\vec{x}_r$ be a vector of variables that occur in the literals in S_r and either in $\vec{x}$ or a literal in S_l. Then, we can rewrite r as:

$$\begin{aligned}
R(\vec{x}) \quad &\text{:-}\ R_l(\vec{x}_l), R_r(\vec{x}_r) \\
R_l(\vec{x}_l) \quad &\text{:-}\ R_{i_1}(\vec{x}_{i_1}), \dots, R_{i_n}(\vec{x}_{i_n}) \\
R_r(\vec{x}_j) \quad &\text{:-}\ R_{j_1}(\vec{x}_{j_1}), \dots, R_{j_m}(\vec{x}_{j_m}),
\end{aligned}$$

where we have $S_l = \{R_{i_1}(\vec{x}_{i_1}), \dots, R_{i_n}(\vec{x}_{i_n})\}$ and $S_r = \{R_{i_1}(\vec{x}_{i_1}), \dots, R_{i_n}(\vec{x}_{i_n})\}$. We can iteratively apply this rewriting rule to normalize the query. Observe that this is a greedy process, and hence minimality of the normal form is not guaranteed. We discuss its implications in Section 7.

Secondly, instead of recreating R_l and R_r at every step of the normalization procedure, we *reuse* the invented predicates. That is, if there are two predicates which are described by syntactically identical rule bodies (up to permutation of variables) that exists in the query, we require only one of them. This allows us to reuse predicates and shrink the search space. If one chooses not to *reuse* the predicates, they will be eventually unified during the generalization step. However, this heuristic allows us to reduce the size of the search space and hence accelerate the synthesis process. Note that this optimization does not compromise the end-to-end guarantee of Theorem 4.1. For the running example, the normalization of $P^{\leq 3}_{\mathsf{scc}}$ generates the query $\overline{Q}$ in Equation 3.

5 PROVENANCE-GUIDED GENERALIZATION

We can use the normalized relational query as a template to constrain the space of candidate queries to only those which can be constructed by unification of predicates. As discussed in the overview, the user may provide a query like $P^{\leq 3}_{\mathsf{scc}}$ (Equation 1) and intend to generalize it to P_{scc} (Equation 2). In the rest of this section, we develop a provenance-guided technique to solve the minimal generalization problem (Problem 3.2) for queries in the normal form. We have named this provenance-guided generalization procedure Generalize and outline it in Algorithm 2.

We can show that Generalize solves the generalization problem (Problem 3.2) for normal queries with the guarantee that the output of Algorithm 2 will have the least number of invented predicates:

Theorem 5.1. *Given a normalized query $\overline{Q}$, input tuples I and negatively labeled output tuples O^-, the query P generated by* Generalize$(\overline{Q}, I, O^-)$ *is a normalized query that subsumes $\overline{Q}$, does not generate tuples in O^-, and has the fewest invented predicates among all such queries.*

The proof of this theorem relies on Theorem 5.3 which ensures that P subsumes $\overline{Q}$, the soundness check in Step 2(c)i of Algorithm 2 that ensures no tuple in O^- are generated, and the fact that

Algorithm 2 Generalize($\overline{Q}, I, O^-$), where $\overline{Q}$ is a normalized query, I is the set of input tuples and O^- is the set of negatively labeled output tuples.

```
(1) Initialize $\phi := \phi_0(\overline{Q})$.
(2) Let $\overline{Q}$ have $K$ invented predicates. Then, for $k = 1, 2, \ldots, K$:
  (a) Let $\mu = $ Generalize($\overline{Q}, k, \phi$).
  (b) If the Generalize procedure fails to find a unification map $\mu$, then break the loop.
  (c) Otherwise, let $P = T(\overline{Q}, \mu)$.
    (i) If $[\![P]\!](I) \cap O^- = \emptyset$, end the procedure and return $P$.
    (ii) Otherwise, for each $t$ in $[\![P]\!](I) \cap O^-$, update:
            $\phi := \phi \wedge$ Constraint($\overline{Q}, \mu, t$).
```

Step 2 of Algorithm 2 searches for the least number of invented predicates incrementally. Therefore, Algorithm 1 solves Problem 3.2 when the input is a normalized query $\overline{Q}$. In cases where the input query is not normalized, we can guarantee subsumption but not minimality. We discuss this in the Section 9. The rest of this section discusses the details of the algorithm.

5.1 Generalization Algorithm

This algorithm approaches generalization as a unification procedure. That is, as explained in the overview, we rewrite the literals in the query. In order to carry out this process, we seek a map μ from the literals using the invented predicates in $\overline{Q}$ to literals using *fresh* invented predicates. For this section, we consider the following query Q_0 that uses three invented predicates R_1, R_2, and R_3:

$$\begin{aligned} \rho_1 &: \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_2(x, y). \\ \rho_2 &: R_1(x, y) \text{ :- } \texttt{edge}(x, y). \\ \rho_3 &: R_2(x, z) \text{ :- } R_1(z, y), R_1(y, x). \\ \rho_4 &: R_3(x, z) \text{ :- } R_1(x, y), R_2(z, y). \end{aligned} \tag{4}$$

Here, the predicate $R_1(x, y)$ represents that there is an edge between x and y, $R_2(x, z)$ represents there is a path of length two from z to x and $R_3(x, z)$ represents there is a path of length three from x to z. Consider a unification map μ that maps all of $R_1(x, y)$, $R_2(x, y)$, and $R_3(x, y)$ to an invented predicate $S_1(x, y)$. Formally, we define:

Definition 5.2 (Unification Map). Given query Q with invented predicates in $\mathcal{R}$ and variables in X, and a set of invented predicates $\mathcal{S}$ that do not occur in Q, a unification map $\mu : \mathcal{R} \cdot X^* \longrightarrow \mathcal{S} \cdot X^*$ is a function from literal $R(\vec{x})$ in Q that use predicate $R \in \mathcal{R}$ to a literal $S(\vec{x}')$ that uses predicate $S \in \mathcal{S}$ and where $\vec{x}'$ is a permutation of variables $\vec{x}$.

In order to apply a unification map μ to a query $\overline{Q}$, we replace each occurrence of the literal $R(\vec{x})$ in $\overline{Q}$ with $S(\vec{x}')$. We denote such a query with $T(\overline{Q}, \mu)$ where T is a transformation that applies μ to $\overline{Q}$. In the running example, we have $T(Q_0, \mu)$:

$$\begin{aligned} \mu(\rho_1) &: \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(x, y). \\ \mu(\rho_2) &: S_1(x, y) \text{ :- } \texttt{edge}(x, y). \\ \mu(\rho_3) &: S_1(x, z) \text{ :- } S_1(z, y), S_1(y, x). \\ \mu(\rho_4) &: S_1(x, z) \text{ :- } S_1(x, y), S_1(z, y). \end{aligned} \tag{5}$$

This method of unification provides a subsumption guarantee that the query $T(Q_0, \mu)$ generates all the tuples generated by Q_0:

Theorem 5.3 (Subsumption). *For every relational query Q and a unification map μ, the query $T(Q, \mu)$ subsumes Q, that is, on every input I, $[\![Q]\!](I) \subseteq [\![T(Q, \mu)]\!](I)$.*

Proof. Consider a tuple t that can be derived by Q and has a derivation tree τ. Then, to prove that t can be derived by $T(Q, \mu)$, we construct a derivation tree for t in $T(Q, \mu)$ by replacing each rule ρ in t by $\mu(\rho)$. It is immediate that the constructed tree uses rules in $T(Q, \mu)$ to derive t. □

However, $T(Q, \mu)$ may derive undesirable tuples, for instance, consider the tuple scc(c, d) derived by $T(Q_0, \mu)$ (as shown in Figure 5a). Hence, $T(Q_0, \mu)$ is an incorrect generalization and we would like to prune it out in the next iteration of the generalization procedure. Observe that for a given query $\overline{Q}$, the space of unification maps is finite. One can enumerate all unification maps, construct the corresponding queries and check if they are consistent with the input-output example. If there are K predicates in $\overline{Q}$ and k predicates after unification, then the number of possible maps are given by:

$$\sum_{k=1}^{K} \left\{ {K \atop k} \right\} (K-k)n! \geq n! \sum_{k=1}^{N} \binom{K}{k-1} (K-k) \backsim n!2^K \geq 2^{n+K},$$

where $\left\{ {K \atop k} \right\}$ is the Stirling number of the second kind and we assume that each predicate has arity n and the signatures are untyped. This implies that number of candidate queries that can be generated by unification grow exponentially in both the number of invented predicates in the normalized query as well as the arity of the predicates, making an exhaustive search infeasible. Therefore we reduce this search problem to a constraint satisfaction problem by encoding the possible unification maps as variables, and prune it using provenance.

5.2 Encoding Generalization as Constraint Satisfaction

Let the bound on the number of invented predicates in the candidate query be k. Then, for every invented predicate R of arity n in $\overline{Q}$, we introduce:

(1) an integer variable $c(R)$ such that $1 \leq c(R) \leq k$, and
(2) for each integer i such that $1 \leq i \leq n$, an integer variable $p(R, i)$ such that $1 \leq p(R, i) \leq n$.

We only permit aliasing between relations R_1 and R_2 of equal arities, n_{R_1} and n_{R_2} respectively. I.e., $n_{R_1} \neq n_{R_2} \implies c(R_1) \neq c(R_2)$. We also require that $p(R, i)$ should be unique for i. That is, for all relations R and indices i and j, $p(R, i) = p(R, j) \implies i = j$.

The conjunction of the above constraints form the initial constraint $\phi_0(\overline{Q})$. In order to interpret an assignment to this encoding as a unification, we say that the literal $R(x_1, \ldots, x_n)$ is mapped to $S_{c(R)}(x_{p(R,1)}, \ldots, x_{p(R,n)})$.

Example 5.4. For the query in Equation 4, we would have:

$$\phi_0(Q_0) = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4,$$

where the first conjunct ϕ_1 ensures that each relation R_i appearing in Q_0 is mapped to some relation in $S_1, S_2, \ldots, S_k$:

$$\phi_1 = 1 \leq c(R_1) \leq k \wedge 1 \leq c(R_2) \leq k.$$

Because both relations R_1 and R_2 have equal arities, there are no restrictions on their potentially being aliased:

$$\phi_2 = \text{True}.$$

The final two conjuncts ϕ_3 and ϕ_4 ensure the schemas of the source and destination relations, R_i and $S_{c(R_i)}$ are permutations of each other:

$$\phi_3 = 1 \leq p(R_1, 1) \leq 2 \wedge 1 \leq p(R_1, 2) \leq 2 \wedge$$

$$1 \le p(R_2, 1) \le 2 \wedge 1 \le p(R_2, 2) \le 2, \text{ and}$$
$$\phi_4 = p(R_1, 1) \ne p(R_1, 2) \wedge p(R_2, 1) \ne p(R_2, 2).$$

5.3 Provenance-Guided Constraint Generation

Observe that μ is an incorrect generalization and we would like to eliminate the assignment that leads to it. For this purpose, we carefully analyze the program $T(Q_0, \mu)$. Intuitively, it is clear that unifying R_1 and R_2 can lead to an incorrect program as the former represents paths and the latter represents reverse paths. Therefore, one can assign the *blame* of incorrect generalization to the unification of $R_1(x, y)$ with $R_2(x, y)$, and this is independent of how $R_3(x, y)$ is unified with either of the two.

In general, the goal is to identify a minimal set of predicates whose unification leads to incorrect generalization, and use this to prune out all unification maps that contain them. For this purpose, we construct a program that is equivalent to $T(Q, \mu)$ by introducing *tunneling clauses* to Q. In the unification map if some predicate $R(\vec{x})$ is unified with $R'(\vec{x}')$, then we add the rules $R(\vec{x})$:- $R'(\vec{x}')$ and $R'(\vec{x}')$:- $R(\vec{x})$. For a query Q and unification map μ, the program constructed using the tunneling clauses is represented as $T'(Q, \mu)$. We wish to show that $T(Q, \mu)$, the program generated by unifying predicates in Q is semantically equivalent to $T'(Q, \mu)$, the program generated by adding the tunneling clauses.

Theorem 5.5 (Tunneling). *The programs $T(Q, \mu)$ and $T'(Q, \mu)$ are semantically equivalent.*

Proof. Consider an input I. We will show that $T(Q, \mu)$ can derive a tuple t using input I, if and only if, $T'(Q, \mu)$ can derive t. The proof in both directions proceed by structural induction on the derivation tree of t.

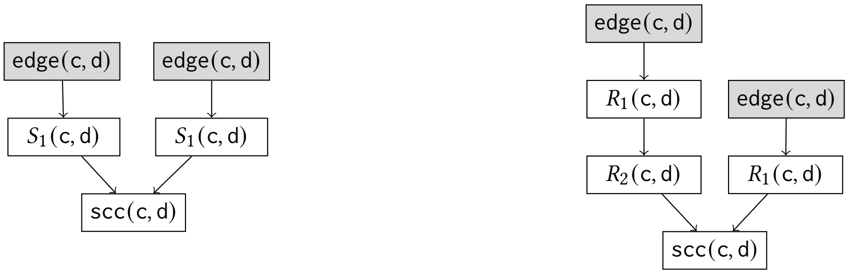
In the forward direction, consider a derivation of t in $T(Q, \mu)$. If $R(\vec{x}')$ is unified to $S(\vec{x})$, for every rule of the form $S(\vec{x})$:- $R_{in}(\vec{x}_{in})$ we introduce the rule $R(\vec{x}')$:- $R_{in}(\vec{x}_{in})$. For every rule $\mu(\rho)$ of the form $S(\vec{x})$:- $S_1(\vec{x}_1), S_2(\vec{x}_2)$, consider the rules $\mu(\rho_1)$ and $\mu(\rho_2)$ whose heads derive the predicates $S_1(\vec{x}_1)$ and $S_2(\vec{x}_2)$. Let ρ be $R(\vec{x})$:- $R_1(\vec{x}_1), R_2(\vec{x}_2)$ and the heads of ρ_1 and ρ_2 be $R'_1(\vec{x}'_1)$ and $R'_2(\vec{x}_2)'$ respectively. As we have $\mu(\rho)$ using the heads of $\mu(\rho_1)$ and $\mu(\rho_2)$, we have that $R_1(\vec{x}_1)$ is unified with $R'_1(\vec{x}'_1)$ to form $S_1(\vec{x}_1)$ (and similarly for $S_1(\vec{x}_1)$). Hence, we can introduce the rules ρ and the tunneling clauses $R_1(\vec{x}_1)$:- $R'_1(\vec{x}'_1)$ and $R'_1(\vec{x}'_1)$:- $R_1(\vec{x}_1)$. The corresponding derivation tree in $T'(Q, \mu)$ can derive t.

Now consider a derivation tree in $T'(Q, \mu)$. If it uses a rule $\rho \in Q$, we introduce the rule $\mu(Q)$. If it uses a tunneling clause $R_1(\vec{x}_1)$:- $R_2(\vec{x}_2)$, then it must be the case that μ unifies $R_1(\vec{x}_1)$ and $R_2(\vec{x}_2)$ to some $S(\vec{x})$. Then, we introduce the rule $S(\vec{x})$:- $S(\vec{x})$. The corresponding tree derives t using rules in $\mu(Q)$ along with tautological rules of the form $S(\vec{x})$:- $S(\vec{x})$. Observe that tautological rules can be eliminated in the derivation tree as their head is the same as the predicate in the premise. This gives us a derivation tree for t using rules $\mu(Q)$.

Therefore, any tuple that can be derived in $\mu(Q)$ can be derived in $T'(Q, \mu)$, and they are semantically equivalent queries. □

Below is the query with tunneling clauses for the running example Q_0 with unification μ (that is, $T(Q_0, \mu)$ as in Equation 5) .

$$\begin{array}{rlrl}
\rho_1 : & \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_2(x, y). & \rho_6 : & R_1(x, y) \text{ :- } R_3(x, y). \\
\rho_2 : & R_1(x, y) \text{ :- } \texttt{edge}(x, y). & \rho_7 : & R_2(x, y) \text{ :- } R_1(x, y). \\
\rho_3 : & R_2(x, z) \text{ :- } R_1(z, y), R_1(y, x). & \rho_8 : & R_2(x, y) \text{ :- } R_3(x, y). \\
\rho_4 : & R_3(x, z) \text{ :- } R_1(x, y), R_2(z, y). & \rho_9 : & R_3(x, y) \text{ :- } R_1(x, y).
\end{array}$$



(a) Example derivation tree of the output tuple scc(c, d) for the query $T(Q_0, \mu)$.

(b) Example derivation tree of the output tuple scc(c, d) for the query $T'(Q_0, \mu)$.

Fig. 5. The derivation tree of the tuple scc(c, d) for the queries $T(Q_0, \mu)$ and $T'(Q_0, \mu)$. The input to the query is the graph of Figure 2a.

$$\rho_5 : R_1(x, y) \text{ :- } R_2(x, y). \qquad\qquad \rho_{10} : R_3(x, y) \text{ :- } R_2(x, y).$$

On evaluating the program $T(Q_0, \mu)$ on the input graph of Figure 2b, we observe that it derives the tuple scc(c, d). By Lemma 5.5, $T'(Q_0, \mu)$ also derives scc(c, d). Figure 5 shows the derivation tree for the two programs.

We will use the derivation tree of scc(c, d) in $T'(Q, \mu)$ to assign the blame of generating a tuple in O^-. That is, in the running example, we analyze the derivation tree in Figure 5b and seek all tunneling rules used in the derivation. Observe that the tree uses only the rule ρ_7 which corresponds to unifying $R_1(x, y)$ and $R_2(x, y)$. Any unification map that unifies these two predicates (with the same rearrangement of variables) will generate scc(c, d) and we can eliminate them in the future iterations. However, the derivation tree does not use a rule with R_3, and hence we can conclude that a unification of R_3 is irrelevant to the derivation of the undesirable tuple. In this sense, the analysis of the derivation tree gives us a part of the unification to assign blame for generating a tuple in O^-.

In general, if the derivation tree includes a tunneling clause of the form $R(\vec{x}_1) \text{ :- } R'(\pi(\vec{x}))$ for some permutation of variables π, we add the constraint:

$$\neg \left(c(R) = c(R') \wedge \bigwedge_{i=1}^{k} p(R, i) = p(R', \pi(i)) \right),$$

where k is the arity of R. This constraint prunes out all unifications where $R(\vec{x}_1)$ is unified with $R'(\pi(\vec{x}))$. If the derivation tree uses more than one tunneling clause, we take the conjunction of all of them.

The process of constructing the derivation tree of a tuple t in $T'(Q, \mu)$ is implemented as a subroutine Constraint(Q, μ, t), and is used in Algorithm 2.

6 EXPERIMENTAL EVALUATION

Our implementation of Möbius consists of approximately 1,300 lines of Python code. We use Soufflé [Zhao et al. 2020] to evaluate candidate queries and compute data provenance, and we use Z3 to solve the constraints generated by the Generalize procedure. Our evaluation in this section attempts to answer the following questions:

Q1. *Effectiveness:* How effective is Möbius in synthesizing queries with a variety of recursion schemes compared to state-of-the-art tools?
Q2. *Generalizability:* Does predicate unification improve accuracy when the learned query is tested on unseen data?
Q3. *Expressibility:* How does the expressive power of Möbius compare against the baselines?
Q4. *Convergence:* Does accounting for data provenance improve convergence time?

We describe our benchmark suite in Section 6.1 and the three baselines against which we compare Möbius in Section 6.2. We present our findings for **Q1**, **Q2**, **Q3**, and **Q4** in Sections 6.3–6.6 respectively.

6.1 Benchmarks

We evaluate Möbius on a suite of 21 synthesis tasks obtained from the domains of knowledge discovery and program analysis. The intended solutions for all of these tasks involve the use of recursion. We present a summary of these benchmarks in Table 1. The benchmarks are divided into seven categories:

(1) *Transitive Closure*: This is the simplest example of a recursive query that constructs the transitive closure of the input predicate. We use the example of reachability in directed graphs for this category.
(2) *Boolean Transitive Closure*: This category comprises of queries that involve transitive closure and some Boolean operation such as conjunction or disjunction. It includes five benchmarks that draw from the domains of knowledge discovery and program reasoning.
(3) *Linear Queries*: A linear query is one where the invented (or output) predicate occurs at most once in each rule [Abiteboul et al. 1995]. While the previous two benchmark categories also include only linear queries, this category includes three benchmarks from knowledge discovery that are not covered by Boolean transitive closure.
(4) *Intersection*: These queries correspond to intersection of linear queries (such as `scc` is an intersection of `path` and its reverse). This category consists of two benchmarks.
(5) *Schema Invention*: The `monochromatic` query corresponds finding monochromatic paths in a vertex colored graph. We discuss it in detail in Section 6.4.
(6) *Non-linear Queries*: These are three other queries from knowledge discovery and program analysis that cannot be expressed as a linear query.
(7) *Mutual Recursion*: This category consists of six linear and non-linear queries involving mutual recursion, that is, they have two or more recursive predicates that call each other.

These benchmarks are collected from previous literature on relational query synthesis and express a diverse range of challenges from across different application domains.

6.2 Baselines

We compare Möbius with three state-of-the-art synthesizers that use different synthesis techniques: GenSynth [Mendelson et al. 2021], which uses an evolutionary search algorithm, and ILASP [Law et al. 2020] and Popper [Cropper and Morel 2021], which are based on constraint solving techniques.

ILASP and Popper model the synthesis problem as a search through a finite space of candidate queries. In order to evaluate them in our setting, we generated candidate rules for each of the 21 benchmarks using instance-specific *mode declarations*. A mode declaration is a syntactic constraint on the candidate queries such as the length of the rule or the number of times a particular relation can occur in its body. In particular, we provide ILASP with the names and signatures of all predicates, including invented predicates, whether they can appear as the head of a clause, and the maximum number of times each predicate can appear in a clause body. In addition, we also provide the

Table 1. Table summarizing benchmark characteristics. We evaluate Möbius on a suite of 21 benchmarks featuring diverse recursion schemes. For each benchmark, we summarize the number of input-output relations and the number of input-output tuples. Ten of these benchmarks use invented predicates.

Name	Brief description	Input		Output		Reference Solution		
		Preds	Tuples	Preds	Tuples	Rules	InvPreds	Literals
transitive closure								
path	graph reachability [Raghothaman et al. 2020]	1	7	1	31	2	0	3
boolean transitive closure								
ancestor	find ancestor in a family tree [Muggleton et al. 2015]	2	8	1	19	4	1	9
connected	unidirectional graph reachability [Mendelson et al. 2021]	1	20	1	104	4	1	5
escape	escape analysis for Java [Si et al. 2018]	4	13	1	6	6	0	11
union-find	equivalence of elements in same set [Si et al. 2018]	3	21	1	36	4	0	6
wikiedits	extract edit history in Wikipedia	4	16	1	7	2	0	8
linear queries								
rsg	reverse-same-generation in family tree [Abiteboul et al. 1995]	3	17	1	11	2	0	4
sgen	same generation in family tree [Abiteboul et al. 1995]	1	7	1	21	2	0	5
zero	checking equality of numbers	2	12	1	38	6	0	16
intersection								
blue-and-green	graph reachability with two colored paths	2	9	1	5	2	0	5
scc	compute SCCs in graph [Raghothaman et al. 2020]	1	10	1	25	3	1	5
schema invention								
monochromatic	monochromatic paths in a vertex colored graph	2	134	1	56	3	1	6
non-linear queries								
andersen	inclusion-based pointer analysis for C [Andersen 1994]	4	7	1	7	4	0	9
dyck	well balanced parentheses	2	10	1	8	3	0	7
modref	mod-ref analysis for Java [Si et al. 2018]	7	18	5	34	10	0	18
mutual recursion								
1-call-site	1-call-site pointer analysis for Java [Whaley and Lam 2004]	7	28	1	4	4	1	10
1-object	1-object-sensitive pointer analysis [Milanova et al. 2002]	9	40	1	4	4	1	11
1-object-1-type	1-type-1-object sensitive analysis [Smaragdakis et al. 2011]	10	48	1	6	5	2	13
1-type	1-type-sensitive pointer analysis [Smaragdakis et al. 2011]	10	42	1	5	4	1	10
2-call-site	2-call-site pointer analysis for Java [Whaley and Lam 2004]	7	30	1	4	4	1	10
buildwall	learn a stable wall strategy [Muggleton et al. 2015]	4	30	1	4	3	1	7

maximum number of variables in each rule. Similarly, we provide Popper with bounds on the number of learned rules, their lengths, and the number of variables which can occur in each rule. We ensure uniformity by running all baselines in single-threaded mode.

For each benchmark query, we recovered the minimum mode declarations required from its reference solution. For example, consider the query:

$$\texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{path}(y, z).$$
$$\texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).$$

From this target query, we would recover the following mode declarations:

```
#modeb(1, edge(var(V), var(V)), (positive)).
#modeb(2, path(var(V), var(V)), (positive)).
#modeh(path(var(V), var(V))).
#maxv(3).
```

These mode declarations specify that edge and path predicates may appear in rule bodies, and also specify the maximum number of times they may be used. Additionally, the head of a rule can only have path or scc predicate, and no rule should use more than 3 variables. That is, the mode declaration implicitly specify that there is only one recursive predicate path. In case of invented predicates, the user must explicitly provide the invented predicate along with its schema.

Lemma 4.2 of [Thakkar et al. 2021] alternatively provides an instance-agnostic technique to derive mode declarations. However, as we will see in our evaluation in Section 6.3, the baseline tools often run out of time with even the more constrained instance-specific settings, thereby rendering this instance-agnostic approach infeasible.

In summary, we make the most favorable case for these baselines by choosing the tightest set of mode declarations that contains the reference solution for the corresponding synthesis task.

Table 2. Table summarizing effectiveness of synthesis. We evaluate Möbius and the three baselines on a suite of 21 benchmarks. GenSynth(1) and GenSynth(32) correspond to executions with one and 32 threads (its default setting) respectively. The remaining tools are run in single-threaded mode. Möbius successfully synthesizes all benchmarks with an average run-time of 23.1 seconds, while GenSynth(32) times out on one benchmark. GenSynth(1), ILASP and Popper time out on 7 benchmarks each. Note that GenSynth and Popper fail to find a solution for 1 and 7 benchmarks respectively.

Name	**Runtime**				
	Möbius	GenSynth(1)	GenSynth(32)	ILASP	Popper
path	<1	<1	1.1	<1	<1
ancestor	**<1**	5.0	3.3	timeout	21.8
connected	**1.2**	-	-	timeout	-
escape	**<1**	4.5	2.5	1.0	-
union-find	**1.0**	2.1	2.8	20.2	timeout
wikiedits	1.5	157.8	4.5	**1.0**	42.1
rsg	**1.8**	39.5	11.7	27.9	3.7
sgen	1.3	6.5	2.5	2.3	**<1**
zero	11.1	timeout	13.3	**3.4**	-
blue-and-green	**1.6**	17.9	7.0	3.1	3.2
scc	**2.2**	4.7	2.4	timeout	-
monochromatic	14.2	**5.1**	5.2	timeout	timeout
andersen	**5.7**	timeout	143.0	554.8	timeout
dyck	133.9	565.9	**6.3**	52.1	-
modref	**275.8**	timeout	352.8	timeout	-
1-call-site	**1.1**	timeout	5.7	timeout	timeout
1-object	**1.4**	753.7	timeout	299.9	timeout
1-object-1-type	20.8	timeout	**2.7**	455.9	24.6
1-type	**3.1**	timeout	224.3	timeout	timeout
2-call-site	**1.2**	timeout	4.9	406.5	timeout
buildwall	**2.3**	44.3	6.0	194.3	-

6.3 Effectiveness

We compared the performance of Möbius against the baselines by running each of them on the benchmarks. We set a uniform timeout of 15 minutes for all tools, and ran the experiments on a desktop workstation with a Ryzen 9 5950X CPU and 128 GB of memory running Linux. We measured the running time of each of these tools on each benchmark. We present the running times in Table 2.

Overall, Möbius consistently produces solutions in the least time, despite requiring lesser guidance than all three baseline tools. Across the 21 benchmarks, on average, Möbius requires 41%, 76%, and 60% of the time needed by GenSynth(1), ILASP and Popper respectively. Although running GenSynth with 32 threads results in a significant speedup over GenSynth(1), it remains slower than Möbius on all but 3 benchmarks. Observe also that Möbius is the only tool which does not timeout on any benchmark.

We note that Möbius solves all but two synthesis tasks in less than 30 seconds. In the two most expensive benchmarks, modref (a program analysis task) and dyck (matching well-parenthesized strings), more than 85% of the final synthesis time is needed by EGS to produce the seed query in Step 1 of Algorithm 1 (Figure 6 shows a breakdown of the time needed for the initial synthesis of non-recursive queries).

Additionally, ILASP and Popper fail to solve 7 and 14 problem instances respectively. For Popper, these failures arise both from running out of time and because of its inability to synthesize invented predicates. In particular, it reports infeasibility for five synthesis tasks, including the SCC query (which requires the invented predicate `path`).

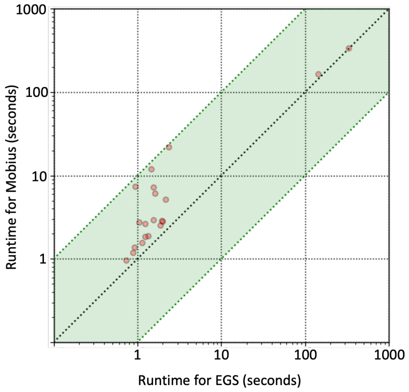


Fig. 6. Comparing the total runtime of the end-to-end Möbius tool against the time spent in the non-recursive phase that uses EGS for the suite of 21 benchmarks.

6.4 Generalizability

Next, we asked whether the generalization algorithm improves the accuracy of learned queries when they are applied to previously unseen datasets. In order to determine the empirical accuracy of Möbius and to compare it to that of EGS and GenSynth, we focused on three synthesis tasks involving graph properties: path, connected, and scc. The two vertices x and y are related by connected(x, y) if there is either a path from x to y or a path from y to x. It is therefore similar to SCC, with the top-level conjunction instead replaced with a disjunction.

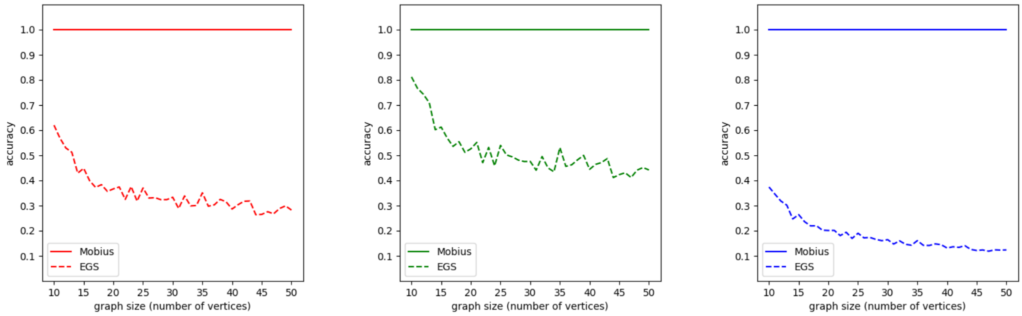
We used the same training data as in Section 6.3. The test data is generated by sampling graphs of increasing size (ranging from 10 to 50 vertices) that contain Hamiltonian cycles. The Hamiltonian cycle ensures that all sampled graphs are non-trivial, i.e., $\emptyset \subsetneq$ path, connected, scc $\subsetneq V \times V$. The experiment is repeated 10 times and the mean values are reported in Figure 7.

The queries learned by EGS only achieve at most 62%, 81%, and 37% test accuracy in each of the three queries. This is unsurprising: because the test graph is larger than any of the training graphs, it is unlikely for non-recursive queries learned by EGS to achieve perfect accuracy in test.

Two expected trends are also evident from this: First, the test accuracy of EGS decreases as the size of the input graph increases. On average, the accuracy falls by over half for graphs with 50 vertices compared to those with 10 vertices. Second, complex benchmarks such as scc that involve invented predicates show significant drops in accuracy compared to relatively simpler benchmarks such as path and connected.

On the other hand, observe that Möbius consistently achieves perfect test accuracy for each of the three benchmarks. Techniques for program synthesis from input-output examples frequently require the training data to be “*representative*”, i.e., adequately describe all behavior modes of the program. In this context, this experiment demonstrates that even when representative training data is provided, the generalization achieved by Algorithm 2 is necessary to learn queries that work on arbitrarily large unseen data.

Another effect of predicate unification is that the learned queries are smaller than the seed non-recursive queries. Over sufficiently large training datasets, the compressed query stops growing once the Generalize procedure has identified the target concept, while the non-recursive query generated by EGS continues to grow as larger instances of these patterns occur in the training data.



(a) Generalization study for the path benchmark.

(b) Generalization study for the connected benchmark.

(c) Generalization study for the scc benchmark.

Fig. 7. Summary of the accuracy studies on three graph benchmarks: path, connected, and scc. Observe that Möbius achieves perfect accuracy on unseen data as recursion is required to express the target concepts.

6.5 Expressibility

As a next example, consider the case of monochromatic which captures monochromatic paths in a vertex colored graph. One can interpret this as finding a connected component of friends with a common characteristic such as having visited the same place or having a shared interest. This benchmark requires *schema invention*, that is, it uses an invented predicate with a schema different from that of the input or output predicates. The following query represents the desired solution:

$$\begin{aligned} \texttt{path}(x, y, c) &\text{ :- } \texttt{edge}(x, y), \texttt{color}(x, c), \texttt{color}(y, c). \\ \texttt{path}(x, z, c) &\text{ :- } \texttt{path}(x, y, c), \texttt{path}(y, z, c). \\ \texttt{monochromatic}(x, y) &\text{ :- } \texttt{path}(x, y, c). \end{aligned}$$

It uses the invented predicate path with arity 3. Notably, path does not share a schema with any of the relations described as part of the input-output data. As discussed before, tools such as ILASP and Popper are limited by requiring the user to specify explicitly specify the invented predicate and its schema. While GenSynth can automatically invent predicates, it assumes that the invented predicate shares its schema with either the input or the output predicates, and therefore fails on benchmarks such as monochromatic which require schema invention.

On the other hand, Möbius can effectively invent the predicate path with arity 3 and synthesize the desired program. This example highlights our two key contributions:

(1) The two-phased approach of first synthesizing a non-recursive query and then generalizing it allows for synthesis of queries with recursive and invented predicates (including those with new schemas), and
(2) The end-to-end synthesis technique is complete (as proved in Theorem 4.1) and does not require additional supervision from the user in terms of mode declarations or schemas for invented predicates.

6.6 Convergence

Finally, we ask whether accounting for data provenance in the generalization process improves convergence time. We construct a variation of Algorithm 2 where the assignment in Step 2(c)ii is

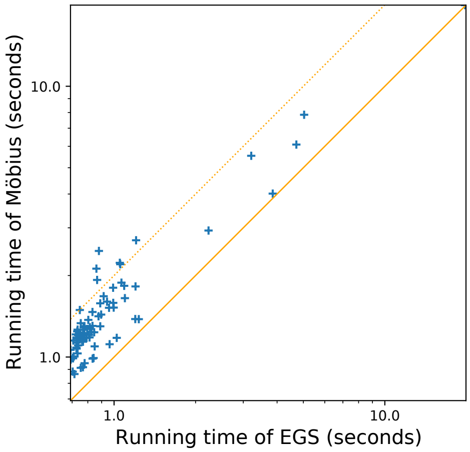


Fig. 8. Comparison of the running time of EGS and Möbius on a suite of 79 non-recursive benchmarks. Because Möbius begins with seed queries produced by EGS, all points are naturally to the top-left of the $y = x$ diagonal. The dotted line corresponds to Möbius taking twice the time needed by EGS. Only 6 benchmarks take longer to complete.

instead replaced by:

$$\phi \coloneqq \phi \wedge \bigvee_{v} (v \neq \mu(v)),$$

where v ranges over all variables currently in context. In other words, we prohibit the constraint solver from producing the same unification map μ in future, but do not perform failure analysis or generalization of any kind.

We ran this modified algorithm on all 20 benchmarks with the same 15 minute time limit as before. In this setting, the algorithm only succeeds on 4 of the 20 synthesis tasks: path, ancestor, union-find, and escape. Observe that all of these are variations of transitive closure. The non-recursive seed queries produced by EGS were correspondingly small and had fewer invented predicates. This greatly reduced the size of the search space, making an exhaustive search feasible. On the other hand, for most other benchmarks, we conclude that provenance-guided generalization is crucial for successful termination.

6.7 Performance on Non-Recursive Benchmarks

The user may occasionally be unaware of whether the intended solution for a problem instance requires the use of recursion. In these cases, they may directly call Möbius in order to synthesize a program, instead of beginning with an exploratory run of a non-recursive query synthesizer. Therefore, in our final experiment, we analyze the performance of Möbius on a suite of 79 non-recursive benchmarks, drawn from the evaluation of EGS in [Thakkar et al. 2021].

We summarize our observations of running time in Figure 8. On average, recursive synthesis imposes only a 57% time overhead, and in all but 6 of the benchmarks, end-to-end synthesis using Möbius requires less than 2× the time needed for synthesis using EGS.

Additionally, it is possible that Möbius generalizes the non-recursive program to a more succinct non-recursive program using an invented predicate. We see this in the case of generating the *grandparent* relation. EGS generates:

```
grandparent(x, z) :- mother(x, y), mother(y, z).
grandparent(x, z) :- mother(x, y), father(y, z).
grandparent(x, z) :- father(x, y), mother(y, z).
grandparent(x, z) :- father(x, y), father(y, z).
```

While this is a correct solution, Möbius generalizes it by inventing a predicate corresponding to the *parent* relation (denoted below with S), and returns the following solution with the size of the program reduced from 8 to 4:

```
                    S(x, y) :- mother(x, y).
                    S(x, y) :- father(x, y).
          grandparent(x, z) :- S(x, y), S(y, z).
```

7 LIMITATIONS AND FUTURE WORK

In this section, we discuss a few limitations of Möbius, and outline opportunities for future work:

(1) *Under-generalization:* In Figure 7, we observe that Möbius has a perfect score of 1.00. However, for more complex benchmarks with small training data, it is possible that the the variety of scenarios covered by the training data may be insufficient to completely convey user intent, resulting into over-fitting queries which are not general enough. As such, this is a well-known limitation of synthesis tools operating in the programming-by-example (PBE) paradigm [Halbert 1984]. One can overcome this limitation by using a richer set of input-output examples that cover all features of the target query.

(2) *Over-generalization:* Conversely, the synthesis algorithm might discover spurious patterns in the training data which allow it to further compress the learned queries. Analogous to the previous limitation, this occurs when the user fails to provide sufficiently many negative examples to preclude overfitting. Providing a representative set of examples can become burdensome in the case of complex concepts. Interactive program synthesis [Le et al. 2017] has the potential to overcome this limitation. Exploring the space of tradeoffs between fully automatic synthesis and the inclusion of a user who actively guides and refines the synthesis process is an exciting direction of future research.

(3) *Greedy normalization:* The normalization strategies can affect the performance of the tool as well as the minimality guarantee from Theorem 5.1. In our implementation, the normalization process greedily generates invented predicates. Based on the choice of these invented predicates, the synthesized query may lead to an over-fitting query, a non-minimal query, or a query that, while equivalent to user intent, may be challenging to interpret due to gratuitous invented predicates. However, note that while the performance of the algorithm depends on the normalization process, the completeness guarantee of Theorem 4.1 ensures that the choice of normal form does not affect the feasibility of the outcome. A potential future direction is to study the impact of alternative normal forms.

(4) *Expressiveness:* A final limitation of our approach concerns the expressiveness of the synthesized queries. As presented, Möbius only supports target queries in *positive relational Datalog* [Abiteboul et al. 1995]. Extending our approach to support additional features including negation, comparison predicates, and aggregation is an important direction of future research.

8 RELATED WORK

We discuss related work on synthesis of relational queries and synthesis of recursive programs.

Synthesis of Relational Queries. The problem of synthesizing recursive relational queries has been explored by some previous approaches. These include constraint-solving techniques like Zaatar [Albarghouthi et al. 2017], ILASP [Law et al. 2020] and Neo [Feng et al. 2018b], enumerative search techniques such as ALPS [Si et al. 2018], hybrid techniques like ProSynth [Raghothaman et al. 2020] and Popper [Cropper and Morel 2021], genetic programming techniques like GenSynth [Mendelson et al. 2021], and neural learning techniques such as NTP [Rocktäschel and Riedel 2017a]. All of these techniques require additional supervision in the form of instance-specific templates to synthesize recursive programs.

We note that techniques like ILASP and Popper target a more expressive fragment of declarative programs. Additionally, neural learning techniques [Dong et al. 2019; Evans and Grefenstette 2018; Rocktäschel and Riedel 2017a; Si et al. 2019; Yang et al. 2017] and GenSynth can handle tasks that involve noise. Neural techniques such as CTPs [Rocktäschel and Riedel 2017b] extend classical techniques to support automated discovery of a minimal set of rules to bias the search.

A number of techniques have been developed to target the non-recursive fragment, such as Scythe [Wang et al. 2017a,c], SQLSynthesizer [Zhang and Sun 2013], and example-guided synthesis [Thakkar et al. 2021]. These techniques do not need instance-specific templates.

Synthesis of Recursive Programs. Beyond the domain of relational queries, a number of program synthesis approaches target recursive programs. Escher is a general purpose algorithm for recursive program synthesis that can be adapted to different domain specific languages [Albarghouthi et al. 2013]. It is parameterized by the components (instructions) that can be used in the program. Synduce uses quantifier bounding for inductive program synthesis where the problem instance is specified as an input reference (recursive) function and a recursion skeleton [Farzan and Nicolet 2021]. Cypress targets the domain of heap-manipulating programs that occur in data structure transformations [Itzhaky et al. 2021], and Burst presents a bottom-up method for synthesizing functional recursive programs from logical specifications.

Generalizability in Program Synthesis. The question of generalizability is important to program synthesis. A number of approaches convert the synthesis problem into an optimization task by requiring to find the simplest program [Gulwani 2011; Mechtaev et al. 2015; Raychev et al. 2016]. Such methods have limited efficiency as requiring the optimal solution greatly increases the difficulty of the problem. The idea of generalization through unification has been studied in the context of conditional linear integer arithmetic (CLIA), however, CLIA is inherently non-recursive [Ji et al. 2021]. There also exist approaches that improve the generalizability by introducing user interactions as input to synthesizers [Ji et al. 2020; Padhi et al. 2018; Wang et al. 2017b].

9 CONCLUSION

We have proposed a novel two-phase method to synthesize recursive relational queries from input-output examples. We first use an example-guided technique to synthesize a seed non-recursive program and then use a constraint-based method to generalize it into a query with recursive and invented predicates. We have implemented this method as an end-to-end synthesis tool named Möbius. The two phased approach allows us to leverage the merits of both the techniques as well as provide theoretical guarantees including completeness and termination. We have evaluated Möbius on a diverse suite of tasks from the literature, and compared it to state-of-the-art synthesizers.

While Möbius targets the domain of relational queries, our technique suggests a way for the fully automated synthesis of recursive programs in different domains such as conditional programs and string transformations. The key idea is the two phased approach of first synthesizing a non-recursive program and then extrapolating and generalizing it by searching for recursion schemes.

ACKNOWLEDGMENTS

We thank the anonymous reviewers whose feedback greatly improved this paper. The research described in this paper was supported by the NSF under grants CCF #2146518, #2124431, and #2107261.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification.* Springer Berlin Heidelberg, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-based synthesis of Datalog programs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP).*

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. DIKU, University of Copenhagen.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009).

Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. 2020. Turning 30: New Ideas in Inductive Logic Programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).*

Andrew Cropper and Rolf Morel. 2021. Learning Programs by Learning from Failures. *Machine Learning* 110 (2021), 801–856.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. 2019. Neural Logic Machines. In *Proceedings of the 7th International Conference on Learning Representations (ICLR).* https://doi.org/10.1145/3340531.3411949

Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Int. Res.* 61, 1 (Jan. 2018), 1–64.

Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification.* Springer International Publishing, 832–855. https://doi.org/10.1007/978-3-030-81685-8_39

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018a. Program Synthesis Using Conflict-Driven Learning. *SIGPLAN Not.* 53, 4 (jun 2018), 420–435. https://doi.org/10.1145/3296979.3192382

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018b. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11.* ACM Press. https://doi.org/10.1145/1926385.1926423

Daniel Conrad Halbert. 1984. *Programming by Example.* Ph. D. Dissertation. AAI8512843.

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM. https://doi.org/10.1145/3453483.3454087

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM. https://doi.org/10.1145/3385412.3386025

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–28. https://doi.org/10.1145/3485544

Mark Law, Alessandra Russo, and Krysia Broda. 2020. The ILASP system for Inductive Learning of Answer Set Programs. *CoRR* abs/2005.00904 (2020).

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *ArXiv* abs/1703.03539 (2017).

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* IEEE. https://doi.org/10.1109/icse.2015.63

Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GenSynth: Synthesizing Datalog Programs without Language Bias. In *Proceedings of the Conference on Artificial Intelligence (AAAI).*

Ana Milanova, Atanas Rountev, and Barbara Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002).* ACM, 1–11.

Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. 2015. Meta-interpretive Learning of Higher-order Dyadic Datalog: Predicate Invention Revisited. *Machine Learning* 100, 1 (01 July 2015), 49–73. https://doi.org/10.1007/s10994-014-5471-y

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–28. https://doi.org/10.1145/3276520

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL).*

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. https://doi.org/10.1145/2837614.2837671

Tim Rocktäschel and Sebastian Riedel. 2017a. End-to-end Differentiable Proving. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Tim Rocktäschel and Sebastian Riedel. 2017b. End-to-end Differentiable Proving. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/b2ab001909a8a6f04b51920306046ce5-Paper.pdf

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided Synthesis of Datalog Programs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3236024.3236034

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog programs using numerical relaxation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Michael Sipser. 2012. *Introduction to the Theory of Computation*. Cengage Learning.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, 17–30. https://doi.org/10.1145/1926385.1926390

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-guided synthesis of relational queries. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017b. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. https://doi.org/10.1145/3035918.3058738

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017c. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

John Whaley and Monica Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004)*. ACM, 131–144. https://doi.org/10.1145/996841.996859

Fan Yang, Zhilin Yang, and William Cohen. 2017. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase.2013.6693082

David Zhao, Pavle Subotic, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020).

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