# Example-Guided Synthesis of Relational Queries

Aalok Thakkar
University of Pennsylvania
Philadelphia, USA
athakkar@cis.upenn.edu

Aaditya Naik
University of Pennsylvania
Philadelphia, USA
asnaik@cis.upenn.edu

Nathaniel Sands
University of Southern California
Los Angeles, USA
njsands@usc.edu

Rajeev Alur
University of Pennsylvania
Philadelphia, USA
alur@cis.upenn.edu

Mayur Naik
University of Pennsylvania
Philadelphia, USA
mhnaik@cis.upenn.edu

Mukund Raghothaman
University of Southern California
Los Angeles, USA
raghotha@usc.edu

## Abstract

Program synthesis tasks are commonly specified via input-output examples. Existing enumerative techniques for such tasks are primarily guided by program syntax and only make indirect use of the examples. We identify a class of synthesis algorithms for programming-by-examples, which we call Example-Guided Synthesis (EGS), that exploits latent structure in the provided examples while generating candidate programs. We present an instance of EGS for the synthesis of relational queries and evaluate it on 86 tasks from three application domains: knowledge discovery, program analysis, and database querying. Our evaluation shows that EGS outperforms state-of-the-art synthesizers based on enumerative search, constraint solving, and hybrid techniques in terms of synthesis time, quality of synthesized programs, and ability to prove unrealizability.

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

***Keywords:*** Programming by example, Example-Guided Synthesis

**ACM Reference Format:**
Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3453483.3454098

## 1 Introduction

Program synthesis aims to automatically synthesize a program that meets user intent. While the user intent is classically described as a correctness specification, synthesizing programs from input-output examples has gained much traction, as evidenced by the many applications of programming-by-example and programming-by-demonstration, such as spreadsheet programming [25], relational query synthesis [51, 57], and data wrangling [19, 33]. Nevertheless, their scalability remains an important challenge, and often hinders their application in the field [5].

Existing synthesis tools predominantly adapt a *syntax-guided* approach to search through the space of candidate programs. While the search does involve evaluating candidate programs on certain inputs, and checking if a program is consistent with the given input-output examples, the examples are used as a black-box, that is, only the *values* of inputs and outputs matter and not how these values are constructed. For instance, the *indistinguishability* optimization [56] considers two expressions equivalent if they produce same outputs on the inputs currently under consideration, and restricts the search to try out only inequivalent subexpressions. It uses the examples to accelerate the search but does not use the structure of the training data to generate candidates. While constraint solving techniques encode the synthesis problem as a set of constraints that encode both the syntactic structure and consistency with the given input-output examples together [2, 32, 48], an off-the-shelf SMT solver is used to solve the resulting constraints with no particular optimization to exploit the structure of examples.

In contrast, PBE-based program synthesizers such as FlashFill [24] examine the structure in the inputs and outputs, and the common patterns between them, to focus the search on candidate programs that are more likely to be consistent with the given examples. Other notable examples of such techniques include FlashRelate [9] and Golem [41]. These techniques are typically more scalable than their purely syntax-guided counterparts. We therefore believe that this is a promising method to address the challenge of scalability, and is worthy of investigation, particularly in contexts where the rich set of syntactic features renders the space of programs too large for syntax-guided approaches. Motivated by these observations, we identify algorithms whose order of program generation depends on latent structure in the training data (as opposed to purely black-box access through an evaluation oracle) as being "*example-guided*".

In this paper, we present an example-guided technique for program synthesis in the setting of relational queries, i.e., declarative logic programs where the input-output examples are tabular relations. Relational queries find applications in domains such as knowledge discovery, program analysis, and in querying databases. Such relational data representations are ubiquitous in practice and form the basis of database query languages such as SQL, Datalog [1], SPARQL [42], Cypher [20], as well as their variants for querying code, such as PQL [37], LogiQL [22], and CodeQL [8].

While this has prompted several recent tools for synthesizing relational queries from examples [32, 47, 57], all of these approaches are syntax-guided. We recognize that there is an opportunity for applying the example-guided approach for the synthesis of relational queries due to the natural correspondence between the elements occurring in the tuples of both the input and output relations.

Our main technical contribution in this paper is to develop an example-guided algorithm to synthesize relational queries. This algorithm, which we call *EGS*, uses co-occurrence patterns in the provided input and output tuples to guide the search through candidate programs. We formalize these co-occurrence patterns by introducing the concept of a *constant co-occurrence graph*, and establish a correspondence between its sub-graphs and the candidate programs being enumerated. As a consequence, the synthesis problem reduces to one of finding a sub-graph with appropriate properties, which the EGS algorithm solves by maintaining a priority queue, in a manner similar to existing syntax-guided algorithms.

At each step, the algorithm evaluates the current candidate program, and incrementally expands it based on its immediate neighborhood in the constant co-occurrence graph. This enables us to prioritize programs based on their accuracy, rather than on purely syntax-guided metrics such as size or likelihood. This prioritization metric is similar to that used in decision tree learning [45], and enables rapid convergence to the target concept. Furthermore, the correspondence between candidate programs and the constant co-occurrence graph guarantees completeness of the EGS algorithm, i.e., when the sub-graph cannot be extended any further, we can prove that there is no program which is consistent with the given input-output examples, thus avoiding divergence of the search process [27].

We have implemented EGS for the fragment of relational algebra queries that involve multi-way joins, unions, and negation. We evaluate it on a suite of 86 tasks from three application domains: knowledge discovery, program analysis, and database queries. We also compare it to three state-of-the-art synthesizers: Scythe [57], which uses enumerative search; ILASP [31], which is based on constraint solving; and ProSynth [47], which uses a hybrid approach. Our experiments demonstrate that EGS outperforms these baselines in synthesis time and the quality of synthesized programs. Moreover, the completeness guarantee of EGS enables it to prove 7 of these tasks as unsolvable. In contrast, Scythe fails to terminate on 5 of these benchmarks, and the syntactic bias mechanisms in ILASP and ProSynth limit their search to a finite space: notably, exhausting this space does not necessarily imply unsolvability.

In summary, this paper makes the following contributions:

1. We identify a category of algorithms for PBE, called Example-Guided Synthesis, which exploit the latent structure in the provided examples while generating candidate programs.
2. We develop an example-guided algorithm named EGS for synthesizing relational queries by leveraging patterns in a data structure called the constant co-occurrence graph, and efficiently enumerating candidate programs using this structure.
3. We show that EGS outperforms state-of-the-art synthesis tools based on enumerative search, constraint solving, and hybrid techniques across multiple dimensions, including running time, quality of synthesized programs, and in proving unsolvability.

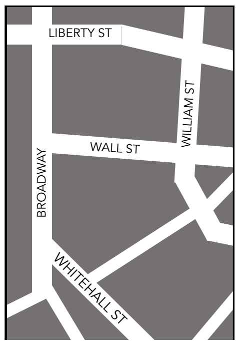
The rest of the paper is organized as follows. Section 2 illustrates the example-guided synthesis approach. Section 3 formulates the relational query synthesis problem. Section 4 presents the core EGS algorithm for synthesizing relational queries. Section 5 extends the core algorithm to support multi-column output relations, union, and negation. Section 6 presents our empirical evaluation. Section 7 discusses related work and Section 8 concludes.

## 2 Overview

We begin by presenting an overview of the example-guided synthesis (EGS) framework. As an example, consider a researcher who has data describing traffic accidents in a city and who wishes to explain this data using information about the road network and traffic conditions.

### 2.1 Problem Setting

We present this data in Figure 1. Suppose that at a given instant, accidents occur on Broadway and Whitehall. The researcher observes that these streets intersect, that they both had traffic, and that the traffic lights on both streets were green. They generalize this observation, and find that the resulting hypothesis, that an accident occurs at every pair of streets with similar conditions, is consistent with the data. One may formally describe their hypothesis as the following Horn clause:



(a)

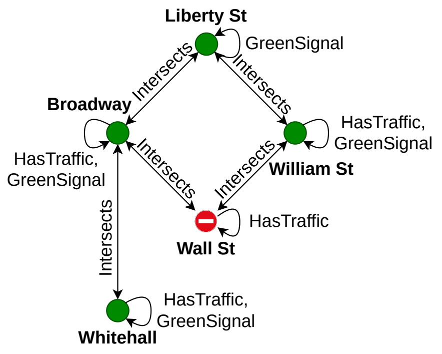
| Intersects | |
|---|---|
| Broadway | Liberty St |
| Broadway | Wall St |
| Broadway | Whitehall |
| Liberty St | Broadway |
| Liberty St | William St |
| Wall St | Broadway |
| Wall St | William St |
| Whitehall | Broadway |
| William St | Liberty St |
| William St | Wall St |

| GreenSignal |
|---|
| Broadway |
| Liberty St |
| William St |
| Whitehall |

| HasTraffic |
|---|
| Broadway |
| Wall St |
| William St |
| Whitehall |

| Crashes |
|---|
| Broadway |
| Whitehall |

(b)



(c)

**Figure 1.** Data describing traffic conditions in a city: (1a) Map of the city, (1b) listing of the input and output relations, and (1c) the induced *constant co-occurrence graph*, $G_I$. We would like to explain the accidents occurring on Broadway and Whitehall.

$$\texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(x, y), \texttt{HasTraffic}(x), \texttt{HasTraffic}(y), \texttt{GreenSignal}(x), \texttt{GreenSignal}(y), \quad (1)$$

where $x$ and $y$ are universally quantified variables ranging over street names, ":-" denotes implication "$\Leftarrow$", and "," denotes conjunction. Our goal in this paper is to automate the discovery of such hypotheses.

This problem can be naturally formalized as a programming-by-examples (PBE) task. Given a set of input facts $I$ encoded as relations, and a set of desirable and undesirable output facts, $O^+$ and $O^-$ respectively, we seek a program which derives all of the tuples in $O^+$ and none of the tuples in $O^-$. In our example, we implicitly assume that the data is completely labelled, so that

$$O^+ = \{ \texttt{Crashes}(\text{Broadway}), \texttt{Crashes}(\text{Whitehall}) \},$$

and $O^-$ is the set of all other streets,

$$O^- = \{ \texttt{Crashes}(\text{Liberty St}), \texttt{Crashes}(\text{Wall St}), \texttt{Crashes}(\text{William St}) \}.$$

Traditional methods for PBE use syntax-guided enumerative techniques that search the space of candidate programs. In our example, a candidate program would be a Horn clause with the premise consisting of one or more of HasTraffic, GreenSignal, or Intersects literals.

A naive approach is to enumerate all candidate programs in order of increasing size till we find a consistent hypothesis. For the running example, we will have to enumerate more than $12 \times 10^6$ candidate programs before discovering the one shown in Equation 1. Unsurprisingly, most work on program synthesis has focused on reducing the size of this search space: in our context, tools such as ALPS and ProSynth restrict the search space by only looking for programs composed of rules from a fixed finite set of candidate rules [47, 52], while ILASP constrains the space through "mode declarations" that bound the number of joins (in our case conjunctions) and the number of variables used [31, 32]. On the other hand, Scythe, a synthesis tool for SQL queries, first finds "abstract" queries that over-approximate the desired output, and then searches for concrete instantiations of these abstract queries that are consistent with the data [57].

## 2.2 Example-Guided Enumeration

At their core, these techniques generate candidates using the syntax of the target concept and do not exploit *patterns* present in the underlying data. Consider the alternative representation of the training data shown in Figure 1c, summarizing input facts $I$. We call this the constant co-occurrence graph $G_I$: every constant is mapped to a vertex, and the edges indicate the presence of a tuple in which the constants occur simultaneously. Now focus on the portion of the graph surrounding Whitehall. Of the 18 tuples present in the data, only 4 tuples refer to this street: GreenSignal(Whitehall), HasTraffic(Whitehall), Intersects(Whitehall, Broadway), and Intersects(Broadway, Whitehall). These tuples suggest the following candidate queries:

$$\begin{aligned} q_1 &: \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \\ q_2 &: \texttt{Crashes}(x) \text{ :- } \texttt{HasTraffic}(x), \\ q_3 &: \texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(x, y), \text{ and} \\ q_4 &: \texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(y, x). \end{aligned}$$

Notice that all four of these queries produce the desirable tuples Crashes(Whitehall) and Crashes(Broadway), but also produce several undesirable tuples: two undesirable tuples by $q_1$ and $q_2$, and three undesirable tuples by $q_3$ and $q_4$ respectively.

Each of these candidate programs can be made more specific by considering sets of tuples. For example, one can extend the set $C_1 = \{ \texttt{GreenSignal}(\text{Whitehall}) \}$ which produces $q_1$ with a new tuple HasTraffic(Whitehall) to obtain:

$$q_5 : \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x). \quad (2)$$

In contrast to $q_1$, this query only produces one undesirable tuple, namely, `Crashes(William St)`.

Instead of directly enumerating candidate programs, the EGS algorithm tracks *enumeration contexts*: Each such context is a set of input tuples obtained from a connected subgraph of the co-occurrence graph $G_I$, and can be generalized into a candidate program by systematically replacing its constants with fresh variables.

Our main insight is that the only tuples which increase the specificity of an enumeration context are those which are directly adjacent to it in the co-occurrence graph. For example, consider context $C_5 = \{$ `GreenSignal(Whitehall)`, `HasTraffic(Whitehall)` $\}$ which produces the query $q_5$ in Equation 2. Observe in Figure 1c that there are exactly two tuples incident on $C_5$: $t$ = `Intersects(Whitehall, Broadway)` and $t'$ = `Intersects(Broadway, Whitehall)`. We conclude that there are exactly two contexts which need to be enumerated as successors to $C_5$, namely: $C_6 = C_5 \cup \{t\}$ and $C_7 = C_5 \cup \{t'\}$. These contexts respectively produce the candidate queries:

$$\begin{aligned} q_6 :\quad & \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x), \\ & \qquad\qquad \texttt{Intersects}(x, y), \text{ and} \\ q_7 :\quad & \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x), \\ & \qquad\qquad \texttt{Intersects}(y, x). \end{aligned}$$

The EGS algorithm repeatedly strengthens the enumeration context $C$ with new tuples until it finds a solution program. For example, after five rounds of iterative strengthening, the context grows to include the tuples:

$$\begin{aligned} C = \{ & \texttt{GreenSignal(Whitehall)}, \texttt{HasTraffic(Whitehall)}, \\ & \texttt{Intersects(Whitehall, Broadway)}, \\ & \texttt{GreenSignal(Broadway)}, \texttt{HasTraffic(Broadway)} \}, \quad (3) \end{aligned}$$

which, when used to explain `Crashes(Whitehall)`, produces the desired solution in Equation 1.

## 2.3 Prioritizing the Enumeration Process

Figure 2 presents the overall architecture of the EGS algorithm. It maintains a set of enumeration contexts, organized as a priority queue, and repeatedly extends each of these contexts with a new tuple, in an example-guided manner. Each enumeration context can be naturally abstracted into a candidate query, as discussed in Section 2.2, and the procedure returns as soon as it finds an explanation which is consistent with the data. The priority function depends on both the size of the candidate program, and its accuracy on the training data, and we formally define it in Section 4.3.

Additionally, because the training data is finite, the co-occurrence graph is also finite, and therefore the EGS algorithm will eventually exhaust the space of enumeration contexts. At this point, Lemma 4.2 guarantees the non-existence of a program which is consistent with the training data, thus proving the completeness of the synthesis procedure.

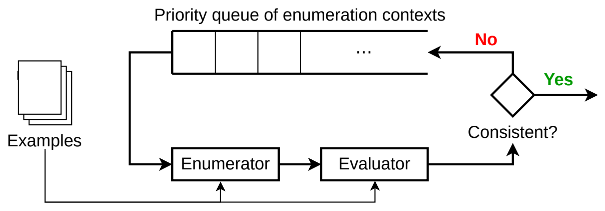


**Figure 2.** Architecture of the EGS algorithm.

While the approach of iteratively strengthening candidate queries is similar to that followed by decision tree learning algorithms [23, 45], a notable difference is the presence of the queue in EGS, which holds alternative candidate explanations. The difference between the two algorithms is therefore similar to the difference between breadth-first search and greedy algorithms, with EGS being biased towards producing small candidate programs.

In our example, a syntax-guided prioritization would be forced to enumerate all programs with less than five joins, which induces an extremely large search space: SCYTHE takes approximately 16 seconds to find a consistent query and ILASP takes approximately 2 seconds, while the EGS algorithm returns in less than one second.

## 2.4 Disjunctions and Multi-Column Outputs

For ease of presentation, the core EGS algorithm that we develop in Section 4 will focus on the case where we have a single desirable tuple with a single field, but with multiple undesirable tuples. Subsequently, in Section 5, we will extend this core algorithm to: (*a*) the case of multiple desirable tuples, by generalizing the space of target concepts to *unions* of conjunctive queries, (*b*) to multi-column output relations by iteratively explaining the fields of the desired tuples, and (*c*) to queries with negation, by pushing the negation operator to the individual literals, and constructing target concepts in negation normal form.

## 3 The Relational Query Synthesis Problem

In this section, we briefly review the syntax and semantics of relational queries and formulate the relational query synthesis problem.

## 3.1 Syntax and Semantics of Relational Queries

A *relational query* $Q$ is a set of Horn clauses. To define their syntax, we start by fixing a set of *input* relation names $\mathcal{I}$ and *output* relation names $\mathcal{O}$. Each relation name $R \in \mathcal{I} \cup \mathcal{O}$ is associated with an *arity* $k$. A *literal*, $R(v_1, v_2, \dots, v_k)$, consists of a $k$-ary relation name $R$ with a list of $k$ variables.

Then, a Horn clause is a rule 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*, or the conclusion which follows from the set of premises, $R_1(\vec{u}_1)$,

$R_2(\vec{u}_2), \ldots, R_n(\vec{u}_n)$, called the *body*. The literals in the body are drawn from $\mathcal{I}$ while $R_h \in \mathcal{O}$. To bound the set of values that each variable may assume, we will follow convention and require that every variable in the head appear at least once in the body.

The semantics of a relational query is interpreted over a data domain $D$ whose elements are called *constants*. For simplicity of formalization, we are assuming that there is a single type. The synthesis framework and its theoretical guarantees can be extended to support typed constants and typed relations.

A *tuple*, $R(c_1, c_2, \ldots, c_k)$, is a $k$-ary relation name $R$ with a list of $k$ constants from $D$. It is an *input* (resp. *output*) tuple if $R \in \mathcal{I}$ (resp. $R \in \mathcal{O}$).

Next, we define *rule instantiation* as follows: Given a map $v$ from variables to the data domain $D$, replace the rule's variables $x$ with constants $v(x)$:

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

For example, consider the query $q_5$ from Equation 2. One can systematically replace its variables according to the map $\{x \mapsto \text{Whitehall}\}$ to obtain the rule instantiation:

$$\texttt{Crashes}(\text{Whitehall}) \impliedby \texttt{GreenSignal}(\text{Whitehall}), \texttt{HasTraffic}(\text{Whitehall}).$$

We say that a tuple $t$ is derivable from input tuples $I$ if there exists a rule $r$ and a map $v$ such that on instantiating $r$ with $v$, the head tuple $R_h(v(\vec{u}_h))$ is $t$, and each of the tuples in the body $R_i(v(\vec{u}_i))$ occur in $I$. Then, a relational query $Q$ takes input tuples $I$ and returns output tuples $O = [\![Q]\!](I)$ as the set of all tuples that are derivable from $I$ using rules in $Q$.

In the literature, each individual rule is also called a *conjunctive query* (CQ), and a set of rules is also called a *union of conjunctive queries* (UCQ). Conjunctive queries are also called *select-project-join* (SPJ) queries because of their representation in relational algebra, and also correspond to queries expressed using the select-from-where idiom in SQL.

In the running example from Section 2, we have $\mathcal{I} = \{\texttt{HasTraffic}, \texttt{GreenSignal}, \texttt{Intersects}\}$, $\mathcal{O} = \{\texttt{Crashes}\}$, and the data domain $D = \{\text{Broadway}, \text{Wall St}, \text{Liberty St}, \text{Whitehall}, \text{William St}\}$. The program in Equation 1 is an example of a conjunctive query.

### 3.2 Problem Formulation

Our ultimate goal is to synthesize relational queries which are consistent with a given set of examples. In this context, an example consists of input and output tuples; the user has labeled the output tuples as either positive or negative. The objective then is to synthesize a program which is consistent with the examples, that is, a program which derives all of the positive tuples and none of the negative tuples.

**Problem 3.1** (Relational Query Synthesis Problem). *Given input relation names $\mathcal{I}$, output relation names $\mathcal{O}$, input tuples $I$, and output tuples partitioned as $O^+$ and $O^-$, return a relational query $Q$ such that $O^+ \subseteq [\![Q]\!](I)$ and $O^- \cap [\![Q]\!](I) = \emptyset$, if such a query exists, and* `unsat` *otherwise.*

We call the triple $M = (I, O^+, O^-)$ an *example*, and a query $Q$ is said to be consistent with it if $O^+ \subseteq [\![Q]\!](I)$ and $O^- \cap [\![Q]\!](I) = \emptyset$.

The user may often be interested in variants of the query synthesis problem. We mention a few such extensions which can be reduced to Problem 3.1:

1. Find a relational query which is simultaneously consistent with multiple input-output examples, $M_1 = (I_1, O_1^+, O_1^-)$, $M_2 = (I_2, O_2^+, O_2^-)$, $\ldots$, $M_p = (I_p, O_p^+, O_p^-)$.
2. Given $I$, and an exhaustively specified set of output tuples $O$, find a program $P$ such that $[\![P]\!](I) = O$.

Our running example has a single input-output example: $(I, \{\text{Broadway}, \text{Whitehall}\}, \{\text{Liberty St}, \text{Wall St}, \text{William St}\})$ where $I$ is described in tables `HasTraffic`, `GreenSignal`, and `Intersects` in Figure 1b. Hereafter, we assume that the data domain $D$ is implicitly specified as the set of all constants that occur in the set of input tuples.

## 4 Example-Guided Synthesis Algorithm

In this section and the next, we formally describe the EGS algorithm for synthesizing relational queries. For ease of presentation, we first develop our core ideas for the case of a single desirable output tuple with a single column, $t = R(c)$. Given a set of input tuples $I$, the target tuple $t$, and a set of undesirable output tuples, the `ExplainCell` algorithm produces a query which is consistent with the example $(I, \{t\}, O^-)$. We extend this synthesis procedure to solve for multi-tuple multi-column output relations in Section 5.

The query is constructed by analyzing patterns of co-occurrence of constants in the examples, which we summarize using the *constant co-occurrence graph*. We first formalize this graph, and then introduce *enumeration contexts* as a mechanism to translate these patterns into relational queries. We conclude the section with a description of the `ExplainCell` procedure which searches for appropriate enumeration contexts using the co-occurrence graph.

### 4.1 The Constant Co-occurrence Graph

Recall that the data domain $D$ is the set of all constants which appear in the input tuples $t \in I$. Then, the *constant co-occurrence graph*, $G_I = (D, E)$, is a graph whose vertices consist of constants in $D$ and with labeled edges $E$ which are defined as:

$$E = \{c_i \xrightarrow{R} c_j \mid \text{input tuple } R(c_1, c_2, \ldots, c_k) \in I\}. \quad (4)$$

In other words, there is an edge $c \to^R c'$ iff there is a tuple $t$ in the input relation $R$ which simultaneously contains both constants $c$ and $c'$. Observe that this makes each edge bi-directional. If constants $c$ and $c'$ occur in a tuple $t$, we say that $t$ *witnesses* the edge $c \rightarrow^R c'$. The constant co-occurrence graph induced by the example of Figure 1b is shown in Figure 1c.

The main insight of this paper is that patterns in the training data can be inferred by examining the co-occurrence relationships between constants. We express these patterns as subgraphs of the co-occurrence graph: as a consequence, the final `ExplainCell` procedure of Algorithm 1 reduces to the problem of enumerating subgraphs of $G_I$.

## 4.2 Enumeration Contexts

An *enumeration context* is a non-empty subset of input tuples, $C \subseteq I$. Equation 3 shows an example of an enumeration context. As Algorithm 1 explores $G_I$, it builds these contexts out of the tuples which witness each subsequent edge.

We can naturally translate a context $C = \{R_1(\vec{c}_1), R_2(\vec{c}_2), \ldots, R_n(\vec{c}_n)\}$ and an output tuple $t = R(\vec{c})$ into a conjunctive query $r_{C \mapsto t}$ as follows:

$$r_{C \mapsto t}: \quad R(\vec{v}) \text{ :- } R_1(\vec{v}_1), R_2(\vec{v}_2), \ldots, R_n(\vec{v}_n), \tag{5}$$

where the head $R(\vec{v})$ and body literals $R_i(\vec{v}_i)$ are obtained by consistently replacing the constants in the output tuple $t = R(\vec{c})$ and in the contributing input tuples $R(\vec{c}_i)$ with fresh variables $v_c$. We say that a context $C$ *explains* a tuple $t$ when the rule $r_{C \mapsto t}$ is consistent with $(I, \{t\}, O^-)$.

Recall from Section 3 that a rule may be instantiated by replacing its variables with constants, analogous to the process of specialization. In contrast, the procedure to obtain $r_{C \to t}$ from the context $C$ and output tuple $t$ may be viewed as a process of generalization. This correspondence between enumeration contexts and rule instantiations allows us to state the following theorem:

**Theorem 4.1.** *Given an example $M = (I, \{t\}, O^-)$, there exists a context $C \subseteq I$ explaining $t$ if and only if there exists a conjunctive query consistent with the example.*

*Proof Sketch.* Clearly, if context $C$ explains $t$, then, by definition, $r_{C \mapsto t}$ is consistent with example $M$. Conversely, if there is a conjunctive query $Q$ consistent with $M$, then let $v$ be a valuation map deriving $t$ in query $Q$. Then, consider the context $C \subseteq I$ to be the set of tuples that occur in the premise of the rule in $Q$ when it is instantiated with $v$. Observe that $r_{C \mapsto t}$ is the rule in query $Q$ and hence the context $C \subseteq I$ explains $t$. □

If a context $C$ explains a tuple $t$ and if $C \subseteq C'$, then $C'$ also explains $t$. We can therefore apply Theorem 4.1 with the largest available context, $C = I$, i.e. the set of *all* input tuples, to prove the following lemma, which establishes the decidability of the relational query synthesis problem:

**Lemma 4.2.** *The given instance of the relational query synthesis problem $M = (I, \{t\}, O^-)$ admits a solution if and only if $r_{I \mapsto t}$ is consistent with $M$.*

## 4.3 Learning Conjunctive Queries

See Algorithm 1 for a description of the `ExplainCell` procedure, which forms the core of the EGS synthesis algorithm. See Figure 2 for a graphical description of its architecture.

The algorithm maintains a priority queue $L$ of enumeration contexts and iteratively expands these contexts by drawing on adjacent tuples from the constant co-occurrence graph $G_I$. It initializes this priority queue in Step 2, with all input tuples $t'$ that contain the target concept $c$. In the case of our running example, to explain the tuple `Crashes(Broadway)`, we would initialize $L$ to $\{C_1, C_2, C_3, C_4\}$, with $C_1 = \{$`GreenSignal(Broadway)`$\}$, $C_2 = \{$`HasTraffic(Broadway)`$\}$, $C_3 = \{$`Intersects(Whitehall, Broadway)`$\}$, and $C_4 = \{$`Intersects(Broadway, Whitehall)`$\}$. These contexts result in the queries $q_1$–$q_4$ shown in Section 2.2. It subsequently iterates over the elements of $L$, and enqueues new contexts for later processing in Step 3(c)ii. In Step 3b, the algorithm returns the first enumeration context which is found to be consistent with the training data.

**Algorithm 1** `ExplainCell`$(I, R(c), O^-)$, where $t = R(c)$ is an output tuple with a single field. Produces an enumeration context $C \subseteq G_I$ such that $r_{C \mapsto t}$ is consistent with the example $(I, \{t\}, O^-)$.

```
1. Let $G_I = (D, E)$ be the constant co-occurrence graph.
2. Initialize the priority queue, $L$:
       $L := \{\{t'\} \mid t' \in I \text{ contains the constant } c\}.$   (6)
   Each element $C \in L$ is a subset of the input tuples, $C \subseteq I$.
3. While $L \neq \emptyset$:
   a. Pick the highest priority element $C \in L$, and remove it from the queue: $L := L \setminus \{C\}$.
   b. If $r_{C \to t}$ is consistent with $(I, \{t\}, O^-)$, then return $C$.
   c. Otherwise:
      i. Let $N = \{c \in D \mid \exists t' \in C \text{ where } t' \text{ contains } c\}$.
      ii. For each constant $c \in N$, edge $e = c \rightarrow^R c'$ in $G_I$, and for each additional input tuple $t' \in I \setminus C$ which witnesses $e$, update:
             $L := L \cup \{C \cup \{t'\}\}.$
4. Now, since $L = \emptyset$, return unsat.
```

A critical aspect of the `ExplainCell` algorithm is the priority function which arranges elements of the queue $L$. The EGS algorithm permits two choices for this priority function: We could consider the enumeration contexts in ascending order of their size, so that:

$$p_1(C) = -|C|.$$

This would guarantee the syntactically smallest solution which is consistent with the data. Alternatively, we could organize the enumeration contexts in lexicographic order of their *scores*, defined as the number of undesirable tuples eliminated per literal,

$$\text{score}(C) = \frac{|O^- \setminus [\![r_{C\mapsto t}]\!](I)|}{|C|},$$

and the size of the context, so that:

$$p_2(C) = (\text{score}(C), -|C|).$$

For example, the score of the contexts $C_1$ from Section 2.2 is 1.0 tuples/literal, as it eliminates one undesirable tuple, `Crashes(Wall St)`, using one literal. On the other hand, the context $C_3$ does not eliminate any undesirable tuples, so that its score is 0. Similarly, the context $C_5$ eliminates two undesirable tuples, `Crashes(Wall St)` and `Crashes(Liberty St)` using two literals, therefore resulting in the score 1.0 tuples/literal. Therefore we have $p_2(C_1) > p_2(C_5) > p_2(C_3)$.

In this way, the priority function $p_2$ simultaneously prioritizes enumeration contexts with high explanatory power and small size, and is inspired by decision tree learning heuristics which greedily choose decision variables to maximize information gain. In practice, we have found this function $p_2$ to result in faster synthesis times than $p_1$ without incurring a significant increase in solution size, and we therefore use this function in our experiments in Section 6. We remark that the solution desired by the user may not always be the smallest conjunctive query which is consistent with the data, and searching for small solutions can sometimes result in overfitting. We further discuss this issue in Section 6.4.

After enumerating all possible contexts, if the algorithm has not found any context which explains the training data, Lemma 4.2 implies that the problem does not admit a solution. The following theorem formalizes this guarantee:

**Theorem 4.3** (Completeness). *Given example $M = (I, \{t\}, O^-)$, where $t = R(c)$,* `ExplainCell`$(I, t, O^-)$ *returns a context $C \subseteq I$ such that the query $r_{C\mapsto t}$ is consistent with $(I, \{t\}, O^-)$ if such a query exists, and returns* `unsat` *otherwise.*

*Proof Sketch.* In the first direction, if `ExplainCell`$(I, t, O^-)$ returns a context $C$ then, by construction, $r_{C\mapsto t}$ is consistent with $(I, \{t\}, O^-)$. To prove the converse, we assume for simplicity that the graph $G_I$ is connected. If `ExplainCell`$(I, t, O^-)$ returns `unsat`, then the last context considered in the loop in Step 3 must have been the set of all input tuples, $C = I$. From Lemma 4.2, it follows that the problem is unsolvable. □

## 5 Extensions of the Synthesis Algorithm

In this section, we extend the central `ExplainCell` procedure described in Algorithm 1 with the ability to synthesize output relations of any arity and with any number of tuples, and also to synthesize queries which require negation.

As an example, we consider the problem of learning kinship relations from the training data in Figure 3. We have two binary (two column) input relations, `father` and `mother`, and

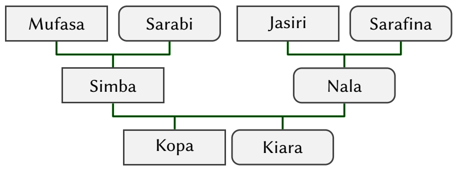


**Figure 3.** Example of a genealogy tree, used as training data to learn the programs $P_{\texttt{grandparent}}$ and $P_{\texttt{sibling}}$. Sarabi, Sarafina, Nala, and Kiara are female while Mufasa, Jasiri, Simba, and Kopa are male.

we would like to learn queries which describe `grandparents` and `siblings`.

### 5.1 Multi-Column Outputs

In order to support multi-column outputs, we explain the fields of the tuple one at a time. Say the output table has $k$ columns, and we wish to explain a tuple of the form $t = R(c_1, c_2, \ldots, c_k)$. We modify the `ExplainCell` procedure to synthesize explanatory contexts $C_1 \subseteq C_2 \subseteq \ldots \subseteq C_k \subseteq I$ such that each context $C_i$ explains the first $i$ fields of $t$, that is, they explain $t[1..i] = R_i(c_1, c_2, \ldots, c_i)$. We call this object the *i-slice* of $t$. We also refer to slices of entire relations such as $O^+[1..i]$ and $O^-[1..i]$ by lifting the slicing operation to sets of tuples in the natural manner.

For example, consider the task of learning the `grandparent` relation from the input data in Figure 3. Consider the output labels:

$$O^+ = \{\texttt{grandparent}(\text{Sarabi}, \text{Kiara})\}$$
$$O^- = \{\texttt{grandparent}(\text{Sarabi}, \text{Simba})\}$$

Then, in order to find a query consistent with $M = (I, O^+, O^-)$, we will first search for a context $C_1 \subseteq I$ which explains $t[1] = \texttt{grandparent}_1(\text{Sarabi})$, and then grow it to $C_2$ which explains $t = t[1..2] = \texttt{grandparent}(\text{Sarabi}, \text{Kiara})$.

Observe that the negative examples also need to be *sliced* appropriately. In this example, the search for a context consistent with $(I, O^+[1], O^-[1])$ would fail since $O^+[1] = O^-[1] = \{\texttt{grandparent}_1(\text{Sarabi})\}$, making this instance unrealizable. We therefore define the *forbidden i-slice*, $F_i$ as the set of tuples $t_f = (c'_1, c'_2, \ldots, c'_i)$ of arity $i$ such that every extension of $t_f$ into a $k$-ary tuple, $t_e = (c'_1, c'_2, \ldots, c'_i, \ldots, c'_k)$, is destined to appear in $O^-$: $t_e \in O^-$. We achieve this by formally defining:

$$F_i = O^-[1..i] \setminus (U \setminus O^-)[1..i], \tag{7}$$

where $U = D^k$ is the set of all $k$-ary tuples over the data domain. In the `grandparent` example we have $F_1 = \emptyset$, resulting in the *sliced* example:

$$M_1 = (I, \{t[1]\}, F_1) = (I, \{\texttt{grandparent}_1(\text{Sarabi})\}, \emptyset).$$

Now, we wish to find $C_1 \subseteq C_2 \subseteq I$ such that $r_{C_1\mapsto t[1]}$ is consistent with $M_1$ and $r_{C_2\mapsto t}$ is consistent with $M$. We can find $C_1$ by calling `ExplainCell`$(I, t[1], F_1)$, which will give us the result:

$$C_1 = \{\texttt{mother}(\text{Sarabi}, \text{Simba})\}.$$

To grow it to $C_2$, we modify the `ExplainCell` procedure to initialize the worklist $L$ in Equation 6 as:

$$\begin{aligned} L &= \{C_1 \cup \{t\} \mid \forall t \in I \text{ containing Kiara}\} \\ &= \{C_1 \cup \{\texttt{father}(\text{Simba}, \text{Kiara})\}, \\ &\qquad C_1 \cup \{\texttt{mother}(\text{Nala}, \text{Kiara})\}\}. \end{aligned}$$

More formally, we define the $\texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$ procedure by modifying the initialization step of Equation 6 so that:

$$L = \{C_{i-1} \cup \{t'\} \mid t' \in I \text{ contains } t[i]\}. \tag{8}$$

We then follow the same process to expand the subgraph one edge at a time, which in case of our running example produces the context:

$$C_2 = \{\texttt{mother}(\text{Sarabi}, \text{Simba}), \texttt{father}(\text{Simba}, \text{Kiara})\}$$

We formally present the `ExplainTuple` procedure in Algorithm 2. The completeness guarantee of Theorem 4.3 carries over as:

**Lemma 5.1.** *Given a context $C_{i-1}$ which explains a sliced example $M_{i-1} = (I, \{t[1...(i-1)]\}, F_{i-1})$, $\texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$ returns a context $C_i \subseteq I$ such that the query $r_{C_i \mapsto t[1..i]}$ is consistent with $M = (I, \{t[1..i]\}, F_i)$ if such a query exists, and returns* `unsat` *otherwise.*

**Algorithm 2** $\texttt{ExplainTuple}(I, t, O^-)$. Given a tuple $t$ with arity $k \geq 1$, synthesizes a context $C$ which is consistent with the example $(I, \{t\}, O^-)$.

```
1. Let $t = R(c_1, c_2, \ldots, c_k)$.
2. Initialize the context $C_0 = \emptyset$.
3. For $i \in \{1, 2, \ldots, k\}$, in order:
   a. Construct the forbidden $i$-slice, $F_i$ as in Equation 7.
   b. Define $C_i = \texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$. If the
      procedure fails, return unsat.
4. Return $C_k$.
```

## 5.2 Unions of Conjunctive Queries

Observe that while the context generated above captures the concept:

$$\texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{father}(z, y),$$

the assumption of a single output tuple does not allow us to express the full grandparent relation (involving both *grandfather* and *grandmother* concepts). We therefore extend the tool to allow for multiple positive output tuples and extend the query language to support disjunctions, that is, we now synthesize unions of conjunctive queries (UCQ). Suppose we are given:

$$\begin{aligned} O^+ = \{&\texttt{grandparent}(\text{Sarabi}, \text{Kiara}), \\ &\texttt{grandparent}(\text{Mufasa}, \text{Kopa}), \\ &\texttt{grandparent}(\text{Jasiri}, \text{Kopa}), \\ &\texttt{grandparent}(\text{Sarafina}, \text{Kiara})\} \\ O^- = \{&\texttt{grandparent}(\text{Mufasa}, \text{Kiara}), \\ &\texttt{grandparent}(\text{Sarafina}, \text{Nala})\} \end{aligned}$$

In order to find a UCQ consistent with $M = (I, O^+, O^-)$, we use a divide-and-conquer strategy: We separately synthesize a conjunctive query that explaining each desired tuple, and then construct their union. Because the rules are non-recursive, it follows that their union is consistent with the training data. In the running example, we get the following queries for each of the four tuples in $O^+$:

$$\begin{aligned} q_1 : &\quad \texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{father}(z, y). \\ q_2 : &\quad \texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{mother}(z, y). \\ q_3 : &\quad \texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{father}(z, y). \\ q_4 : &\quad \texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{mother}(z, y). \end{aligned}$$

Observe that the UCQ with the rules $\{q_1, q_2, q_3, q_4\}$ is consistent with $(I, O^+, O^-)$. This approach is similar to the technique used by EUSOLVER which first synthesizes small programs that conform to portions of the full specification, and then glues them together using conditional statements and case splitting operators provided by the target language [56].

In order to implement this procedure, we maintain a set of unexplained output tuples $O^?$, which is initialized to $O^+$, and repeatedly generate conjunctive queries explaining tuples $t \in O^?$ until all tuples are explained. We construct these conjunctive queries by invoking the `ExplainTuple` procedure of Section 5.1. We formally describe this process in Algorithm 3. Using the completeness guarantee of the `ExplainTuple` procedure, we have:

**Lemma 5.2.** *Given example $M = (I, O^+, O^-)$,* $\texttt{LearnUCQ}(I, O^+, O^-)$ *returns a* union of conjunctive queries $Q$ *consistent with $M$, if such a query exists, and returns* `unsat` *otherwise.*

## 5.3 Negation

Finally, we extend the EGS algorithm to synthesize queries with negation. Similar to propositional formulas, UCQs also admit *negation normal forms*, where the negation operators are pushed down all the way to the individual literals. For example, a rule of the form:

$$r : \quad R(x, y, z) \text{ :- } \neg(R_1(x), R_2(y)), R_3(z).$$

can instead be written as the disjunction of two rules:

$$\begin{aligned} r_1 : &\quad R(x, y, z) \text{ :- } \neg R_1(x), R_3(z). \\ r_2 : &\quad R(x, y, z) \text{ :- } \neg R_2(y), R_3(z). \end{aligned}$$

We therefore limit ourselves to learning UCQs in negation normal form. In our implementation, the user identifies input relation names that can possibly be negated in the final result. For an input relation name $R$ of arity $k$, let $I(R)$ denote the

**Algorithm 3** EGS$(I, O^+, O^-)$. Given an example $M = (I, O^+, O^-)$, finds a UCQ $Q$ consistent with $M$ if such a query exists, and returns `unsat` otherwise.

```
1. Initialize $Q$ to be the empty query, $Q := \emptyset$.
2. Initialize the set of still-unexplained tuples, $O^? := O^+$.
3. While $O^?$ is non-empty:
   a. Pick an arbitrary tuple $t \in O^?$.
   b. Synthesize an explanation,
          $C_t = \texttt{ExplainTuple}(I, t, O^-)$,
      and construct $q_t = r_{C_t \to t}$.
   c. If synthesis fails, return unsat.
   d. Otherwise, update:
          $Q := Q \cup \{q_t\}$, and $O^? := O^? \setminus [\![ q_t ]\!](I)$.
4. Return $Q$.
```

set of tuples in $I$ labeled with $R$. Given the data domain $D$, we explicitly construct the negated relation $\neg R$ with the following tuples:

$$I(\neg R) = \{R(\vec{c}) \mid \vec{c} \in D^k \text{ and } R(\vec{c}) \notin I(R)\}.$$

We add $\neg R$ to the set of input relations and find a solution using Algorithm 3, exactly as before.

Consider, for example the task to learn the sibling relation from the training data in Figure 3. Suppose we are given:

$$O^+ = \{\texttt{sibling}(\text{Kopa}, \text{Kiara})\}$$
$$O^- = \{\texttt{sibling}(\text{Kopa}, \text{Kopa})\}.$$

We can show that no strictly positive program exists which can distinguish the tuples `sibling`(Kopa, Kiara) and `sibling` (Kopa, Kopa) as our hypothesis space does not support the inequality check, Kopa $\neq$ Kiara. If we allow negation, a query consistent with $(I, O^+, O^-)$ is:

$$\texttt{sibling}(x, y) \text{ :- } \texttt{mother}(z, x), \texttt{mother}(z, y), \neg(x = y).$$

We can encode the relation $\neg(x = y)$ using a two-column relation table that pairs unequal constants. We call this relation `neq`, and define it as:

$$I(\texttt{neq}) = \{(c, c') \in D^2 \mid c \neq c'\}.$$

With this additional input relation, EGS is able to solve for the desired concept in less than one second.

## 6 Experimental Evaluation

We have implemented the EGS algorithm in Scala comprising 2200 lines of code. We have provided the code as supplementary material with this paper, and will release it as open-source. In this section, we evaluate it to answer the following questions:

**Q1. Performance:** How effective is EGS on synthesis tasks from different domains in terms of synthesis time?

**Q2. Quality of Programs:** How do the programs synthesized by EGS measure qualitatively?

**Q3. Unrealizability:** How does EGS perform on synthesis tasks that do not admit a solution?

We present our benchmark suite in Section 6.1 and the three baselines against which we compare EGS in Section 6.2. We present our empirical findings for **Q1–Q3** in Sections 6.3–6.5.

We performed all experiments on a server running Ubuntu 18.04 LTS over the Linux kernel version 4.15.0. The server was equipped with an 18 core, 36 thread Xeon Gold 6154 CPU running at 3 GHz and with 394 GB of RAM. Note that EGS is single-threaded and is CPU-bound rather than memory-bound on all benchmarks. Therefore, similar results should be obtained on laptops and desktop workstations with similarly-clocked processors.

### 6.1 Benchmark Suite

We evaluate the EGS algorithm on a suite of 86 synthesis tasks. Of these, 79 admit a solution, meaning there exists a relational query which can perfectly explain their input-output examples. These 79 benchmarks are from three different domains: (*a*) knowledge discovery, (*b*) program analysis, and (*c*) database queries.

*Knowledge discovery.* These benchmarks comprise 20 tasks that involve synthesizing conjunctive queries and unions of conjunctive queries frequently used in the artificial intelligence and database literature.

*Program analysis.* These benchmarks comprise 18 tasks that involve synthesizing static analysis algorithms for imperative and object-oriented programs.

*Database queries.* These benchmarks comprise 41 tasks that involve synthesizing database queries. These tasks, originally from StackOverflow posts and textbook examples, are obtained from Scythe's benchmark suite [57].

There are seven additional benchmarks that do not admit a solution. We describe them in Section 6.5.

Table 1 presents characteristics of all 86 benchmarks, including the number of input-output relations, number of input-output tuples, and whether the intended programs involve disjunctions (∨) or negations (¬). In total, 17 tasks involve disjunctions while 9 of them involve negations. For each benchmark, we provide an exhaustive set of positive output tuples. The tuples not in the positive set are implicitly labelled as negative. This data is provided upfront and not in an interactive fashion. Following our problem setup in Section 3.1, the programs synthesized by EGS do not contain constants. However, some of our benchmarks, such as `adjacent-to-red`, require distinguished constants, such as the rule `target`$(x)$ :- `edge`$(x, y)$, `color`$(y,$ `red`$)$, which references the color `red`. In these cases, we provide an additional input table `isRed`$(x)$ containing a single tuple (`red`) that EGS can use to synthesize the query.

**Table 1.** Benchmark characteristics. For each benchmark, we summarize the number of input-output relations, number of input-output tuples, and whether the intended programs involve disjunctions (∨) or negations (¬).

| Name | Brief description | Input #Relations | Input #Tuples | Output #Relations | Output #Tuples | Features |
|---|---|---|---|---|---|---|
| *Knowledge Discovery* | | | | | | |
| abduce | learn relation induced by abduction [40] | 2 | 12 | 1 | 8 | ∨ |
| adjacent-to-red | identify neighbors of red vertices [14] | 4 | 18 | 1 | 4 | |
| agent | discover strategy for agents on a map [31] | 4 | 106 | 1 | 5 | ¬ |
| animals | distinguishing animal classes [40] | 9 | 50 | 4 | 17 | |
| cliquer | compute 2-paths [47] | 1 | 4 | 1 | 4 | |
| contains | identify allergens in school lunches [50] | 2 | 14 | 1 | 4 | |
| grandparent | discover grandparents in a family tree [14] | 2 | 8 | 1 | 7 | ¬ |
| graph-coloring | identify incorrect vertex-coloring [14] | 2 | 19 | 1 | 3 | |
| headquarters | infer the state of headquarters [50] | 2 | 9 | 1 | 4 | |
| inflammation | bladder inflammation diagnosis [15] | 12 | 640 | 1 | 49 | ∨, ¬ |
| kinship | infer kinship in a family tree [14] | 2 | 8 | 1 | 5 | ∨ |
| predecessor | learn predecessor relation on integers [14] | 1 | 9 | 1 | 9 | |
| reduce | infer symptoms reduced by common drugs [50] | 2 | 10 | 1 | 6 | |
| scheduling | identify conflicts in a schedule [31] | 2 | 8 | 1 | 1 | ¬ |
| sequential | learn 3 generations of ancestory [14] | 2 | 9 | 3 | 17 | ∨ |
| ship | pair products with customers [47] | 3 | 15 | 1 | 5 | |
| son | identify the sons in a family tree [14] | 3 | 12 | 1 | 3 | |
| traffic | explain traffic collision [47] | 3 | 18 | 1 | 2 | |
| trains | distinguish train classes [14] | 12 | 223 | 1 | 5 | |
| undirected-edge | construct symmetric closure [14] | 1 | 3 | 1 | 5 | ∨ |
| *Program Analysis* | | | | | | |
| arithmetic-error | analysis of division by zero errors [35] | 3 | 11 | 1 | 1 | |
| block-succ | basic block analysis | 3 | 21 | 1 | 1 | |
| callsize | memory allocation buffer checker for C | 3 | 21 | 1 | 3 | |
| cast-immutable | type casting checker for Java | 3 | 15 | 1 | 2 | |
| downcast | downcast safety checker for Java [52] | 5 | 89 | 4 | 175 | ¬ |
| increment-float | float increment checker for C | 4 | 16 | 1 | 1 | |
| int-field | integer lattice for points-to-analysis [35] | 3 | 9 | 1 | 1 | |
| modifies-global | identify functions that modify global variables | 3 | 9 | 1 | 1 | |
| mutual-recursion | identify mutual recursion | 1 | 13 | 1 | 3 | |
| nested-loops | infer nested loops with same variable | 3 | 39 | 1 | 3 | |
| overrides | infer overriding in Java | 2 | 6 | 1 | 1 | |
| polysite | polymorphic call-site inference for Java [52] | 3 | 97 | 3 | 27 | |
| pyfunc-mutable | identify mutable arguments to Python functions | 3 | 19 | 1 | 2 | |
| reach | step reachability in dataflow analysis | 3 | 17 | 1 | 2 | |
| reaching-def | reaching definition analysis | 2 | 6 | 1 | 1 | |
| realloc-misuse | memory reallocation checker for C | 3 | 18 | 1 | 1 | |
| rvcheck | return-value-checker in APISan [62] | 4 | 74 | 1 | 2 | |
| shadowed-var | variable shadowing analysis in Javascript | 2 | 12 | 1 | 1 | |
| *Database Queries* | | | | | | |
| sql 1 ∼ 41 | 41 SQL queries [57] | ≤ 6 | ≤ 65 | 1 | ≤ 20 | ∨, ¬ |
| *Unsynthesizable Benchmarks* | | | | | | |
| isomorphism | differentiate isomorphic vertices in a graph | 1 | 2 | 1 | 1 | - |
| sql 42 ∼ 44 | 3 unsynthesizable SQL queries [57] | ≤ 2 | ≤ 8 | 1 | ≤ 4 | - |
| traffic-extra-output | traffic benchmark with extra output tuple | 3 | 18 | 1 | 3 | - |
| traffic-missing-input | traffic benchmark without intersects relation | 2 | 8 | 1 | 2 | - |
| traffic-partial | traffic benchmark with partial input-output | 3 | 11 | 1 | 1 | - |

## 6.2 Baselines

We compare EGS with three state-of-the-art synthesizers that use different synthesis techniques: Scythe [57], which uses enumerative search; ILASP [32], which is based on constraint solving; and ProSynth [47], which uses a hybrid approach by combining search with constraint solving.

ILASP and ProSynth phrase the synthesis problem as a search through a finite space of candidate rules. In order to evaluate them on our benchmark suite, we generated candidate rules for each benchmark using *mode declarations* in ILASP. A mode declaration is a syntactic restriction on the candidate rules such as the length of the rule, number of times a particular relation can occur, and the number of distinct variables used. In our experiments we only focus on the number of times an input relation occurs in a rule, and the number of distinct variables used. Providing a suitable set of mode declarations is a delicate balancing act: generous mode declarations can hurt scalability while insufficient mode declarations can result in insufficient candidate rules to synthesize the desired program. Given a query, one can recover the minimum mode declarations required to generate it. For example, for the program in Equation 1 in the running example, we have the mode declarations:

```
#modeb(2, GreenSignal(var(V)), (positive)).
#modeb(2, HasTraffic(var(V)), (positive)).
#modeb(1, Intersects(var(V),var(V)), (positive)).
#modeh(Crashes(var(V))).
#maxv(2).
```

This specifies for each candidate rule the output relation is `Crashes`, the input relations `GreenSignal` and `HasTraffic` occur at most twice, `Intersects` occurs at most once, and at most two distinct variables are used. This particular choice of modes generates 97 rules. Increasing the mode declarations results in a larger space of candidate rules. For our suite of benchmarks, we observed that a given input relation occurs in a rule at most thrice (such as in `sequential`), and the number of distinct variables in a single rule are at most 10 (as in `increment-float`). This allowed us to generate two set of candidate rules per benchmark:

1. *Task-Agnostic Rule Set*: Candidate rules where any given input relation occurs at most thrice and the number of distinct variables is at most 10, and
2. *Task-Specific Rule Set*: Candidate rules generated using the minimum mode declarations for the desired program.

With a threshold of 300 seconds, the candidate rule enumerator timed out when generating the task-agnostic rule set for 31 of the 79 benchmarks and the task-specific rule set for 2 benchmarks. We summarize the number of candidate rules generated per benchmark in Appendix A.

Similar to EGS, Scythe does not require a set of candidate rules, but the fragment of relational queries targeted by Scythe is SQL (with selection, join, projection, constant comparison, aggregation, and union). In order to compare the four tools fairly, we disable Scythe's support for aggregations. Also, Scythe supports complete labeling, that is every tuple either occurs in $O^+$ or $O^-$; therefore, we consider the set of negative examples $O^-$ to be all tuples of appropriate arity that do not occur in $O^+$.

## 6.3 Q1: Performance

We ran EGS and the three baselines (with ProSynth and ILASP with two sets of candidate rules each) on all 79 benchmarks with a timeout of 300 seconds. We tabulate the results in Appendix A, and present a graphical summary in Figure 4.

EGS runs fastest with an average runtime of under a second and no timeouts. In fact, for all but 6 benchmarks, EGS

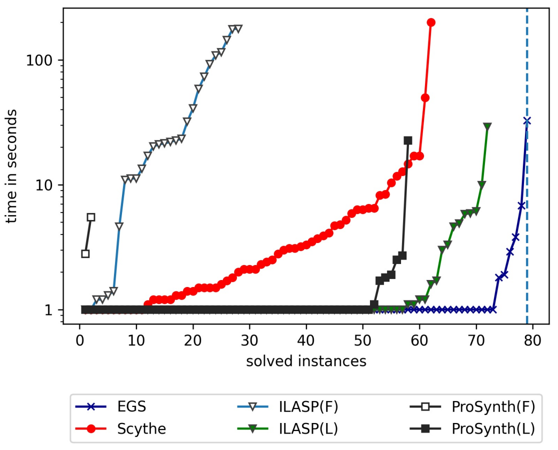


**Figure 4.** Results of our experiments using EGS, Scythe, ILASP, and ProSynth to solve a suite of 79 benchmarks. A datapoint $(n, t)$ for a particular tool indicates that it solved $n$ of the benchmarks in less than $t$ time. Note EGS was the only tool to solve all 79 benchmarks. L and F refer to Task-Specific and Task Agnostic Rule Sets respectively.

returns a solution in less than one second, and never takes more than 33 seconds for any benchmark. Scythe takes an average of 7.6 seconds for 62 benchmarks and times out on 17 of the rest.

When provided with a task-specific rule set, both ILASP and ProSynth exhibit competitive performance on a subset of the benchmarks, and return a solution in less than one second for 57 and 51 benchmarks respectively. Still, their performance suffers on the more complicated benchmarks, and they exhibit timeouts on 7 and 21 of the 79 benchmarks, respectively. However, when provided with a task-agnostic rule set, the performance of both tools quickly degrades, and they timeout on 51 and 77 benchmarks, respectively.

All three baselines are disadvantaged by the enumeration required, and this causes EGS to outperform them, especially on benchmarks with larger numbers of input tuples, larger numbers of relations, or complex target queries. ProSynth and ILASP sometimes outperform EGS when provided with a task-specific choice of target rules on particularly simple benchmarks. However, we emphasize that, in all these cases, all three tools solve the problem in less than one second.

Notably, there are four benchmarks where EGS succeeds, but where all other tools time out: `animals`, `sequential`, `downcast`, and `polysite`. Upon examination, these benchmarks reveal the situations which cause the baseline techniques to underperform. For example, the `animals` benchmark involves classifying animals into their taxonomic classes based on their characteristics which are represented through 9 input relations. The larger number of input relations induces a complex search space causing Scythe to timeout. Furthermore, ILASP enumerates over 2000 candidate rules, even in the task-specific setting, causing both ILASP and ProSynth to also timeout.

## 6.4 Q2: Quality of Programs

We investigated the quality of the synthesized programs for each of the 79 benchmarks and observed that the program synthesized by EGS captures the target concept. For all but two cases, the programs generated by EGS also matched a program crafted by a human programmer. The two outliers are `sequential` and `sql36`. In `sequential`, one of the tasks is to learn the `great-grandparent` relation. The desired program has eight rules (each representing a combination of the `mother` and `father` input relations to form rules of size three); however, we are provided with only two output tuples, and hence we learn a program with 2 rules that correctly explains the data. This can be fixed by adding more training data such that it covers all cases of the target concept. In case of `sql36`, the task involves comparing numbers; however, the input only includes the successor relation. The output of EGS therefore unfolds the greater-than relation using a four-way join of successors. While this is the smallest query that one can generate consistent with the examples, a more succinct query can be learned if we are provided an input table for the greater-than relation. In general, we observe overfitting when either there exists a program consistent with the input-output examples that is smaller than the desired program (as in the case of `sequential`) or when the training data does not represent all of the desired features of the target program (as in `sql36`). In general, one can overcome these cases by providing a richer set input-output examples.

One may also observe overfitting when our heuristic generates a consistent but larger program. This is possible as the priority function greedily optimizes over explanatory power and size simultaneously. We have not observed this case in any of our 78 benchmarks.

We also manually inspected the outputs of the baselines. The programs synthesized by ProSynth and ILASP are identical to ours in the cases when the tools terminate (in both, task-agnostic and task-specific rule sets). However, the programs synthesized by Scythe are neither small nor easy to generalize. In many cases, including knowledge discovery benchmarks such as `adjacent-to-red`, `graph-coloring`, and `scheduling`, we find the synthesized queries to be inscrutable.

## 6.5 Q3: Unrealizability

To test the completeness guarantees provided by the EGS algorithm, we evaluated it on 7 unrealizable benchmarks. The results of these experiments are summarized in Table 2.

The first benchmark, `isomorphism`, is the simplest benchmark which does not admit a solution. In this benchmark, we have the input $I = \{\texttt{edge}(\texttt{a}, \texttt{b}), \texttt{edge}(\texttt{b}, \texttt{a})\}$, and attempt to distinguish between the two vertices by specifying the

**Table 2.** Unrealizable benchmarks. For each benchmark, we summarize runtimes on EGS and the three baselines. Note that Scythe overfits `sql42` and `traffic-partial` using operators like comparisons and negation.

| # Benchmark | EGS | Scythe | ILASP | ProSynth |
|---|---|---|---|---|
| `isomorphism` | 0.1 | - | 0.2 | 12.4 |
| `sql42` | 0.2 | 1.79 | 0.6 | - |
| `sql43` | 0.1 | - | - | - |
| `sql44` | 0.1 | - | - | - |
| `traffic-extra-output` | 0.2 | - | 0.2 | 0.1 |
| `traffic-missing-input` | 0.1 | - | 0.1 | 0.4 |
| `traffic-partial` | 59.5 | 2.33 | 0.2 | 1.5 |

outputs, $O^+ = \{\texttt{a}\}$ and $O^- = \{\texttt{b}\}$. From symmetry considerations, it follows that the benchmark does not admit a solution, and our algorithm successfully reports this in less than one second, while Scythe times out on this benchmark, and ILASP and ProSynth claim that there is no solution with respect to the given mode declarations.

We remark that while ILASP and ProSynth do not provide completeness guarantees like we do, Lemma 4.2 allows us to also strengthen their claims. Observe that as the input $I$ has only two tuples, and any rule explaining the tuple needs at most one join. This can be used to construct an upper bound on the mode declaration which permits ILASP and ProSynth to also prove the unrealizability of the benchmark. However, as these mode declarations grow with the set $I$, we observe time outs in other unrealizable benchmarks.

The next three benchmarks `sql42`–`sql44` are sourced from the Scythe's benchmark suite, and involve some form of aggregation, which is unsupported by EGS. The task in `sql42` is to assign row numbers to the tuples, in `sql43` is to get the top two records grouped by a given parameter, and in `sql44` is to sum items using several IDs from another table. EGS proves the unrealizability of each of these tasks in less than a second. For `sql42`, Scythe produces an overfitting solution (using comparison operators) and ILASP proves the absence of a solution in less than a second. The mode declarations for these benchmarks were the same as that for the task-agnostic rule set.

The final three unrealizable benchmarks are modifications of the running example generated by adding noise. In `traffic-extra-output` we have a constant in the output that does not occur in the input, in `traffic-missing-input` we do not provide the `Intersects` input relation, and in `traffic-partial` we remove certain input and output tuples which are essential to explain the crashes. While Scythe overfits a solution to `traffic-partial` using negation, EGS takes about a minute to prove that there cannot exist a solution which does not involve negation or aggregations.

## 7 Related Work

We discuss related work on program synthesis frameworks, synthesis of logic programs, and example-guided search.

***Program synthesis frameworks.*** General frameworks have been proposed to specify program synthesis tasks. SyGuS [3] formulates program synthesis as a computational problem whose target is specified by a logical constraint and a syntactic template. SKETCH [53, 54] allows the programmer to specify the synthesis task via a syntactic sketch in high-level languages like C and Java. Rosette [55] extends Racket with language constructs for program synthesis and compiles it to logical constraints that are solved using SMT solvers. PROSE [44] provides APIs to synthesize a ranked set of programs that satisfy input-output examples.

Synthesis techniques underlying these frameworks are typically based either on search or constraint solving. Search-based techniques follow the counterexample-guided inductive synthesis (CEGIS) [54] paradigm which combines a search algorithm with a verification oracle. They use examples to implement a number of optimizations such as the indistinguishability optimization to accelerate search [56, 57], *divide-and-conquer* strategies to complete enumerated sketches [19, 28, 57], and *probabilistic models* of programs to bias the search [17, 34, 38]. On the other hand, these techniques are more broadly applicable, and EGS is not directly extensible to domains beyond relational queries.

Lastly, the idea of explaining different tuples by different rules, yielding a union of conjunctive queries, is reminiscent of the search technique in EUSOLVER [4], which employs a divide-and-conquer approach by separately enumerating (a) smaller expressions that are correct on subsets of inputs, and (b) predicates that distinguish these subsets. These expressions and predicates are then combined using decision trees to obtain an expression that is correct on all inputs. Akin to our approach, EUSolver also uses information-gain based heuristics to learn compact decision trees.

***Synthesis of logic programs.*** There is a large body of work on synthesizing logic programs from examples [13]. Existing approaches to this problem are broadly classified into Inductive Logic Programming (ILP), e.g. Metagol [39]; Answer Set Programming (ASP), e.g. ILASP [30]; program synthesis, e.g. ProSynth [47] and Scythe [57]; and neural learning, e.g. NTP [49].

Several of these approaches consider more general program classes than relational queries, but fundamentally differ from EGS in two respects: they are syntax-guided—and therefore require various forms of language bias mechanisms upfront, such as templates (Metagol), mode declarations (ILASP), and candidate rules (ProSynth)—and they do not provide completeness guarantees.

Even bottom-up ILP algorithms that start with examples require language biasing. These approaches use Plotkin's least-general generalisation [43] to produce the most specific clause that generalises the given examples. However, the most specific clause can grow unboundedly and tools such as Golem [41] use restrictions on the background knowledge and hypothesis language to synthesize smaller programs.

Neural learning [16, 18, 49, 61] can handle tasks that involve noise or require sub-symbolic reasoning. NeuralLP [61], NLM [16], and ∂ILP [18] model relation joins as a form of matrix multiplication, which limits them to binary relations. NTP [49] constructs a neural network as a learnable proof (or derivation) for each output tuple up to a predefined depth (e.g. $\leq 2$) with a few (e.g. $\leq 4$) templates, where the network could be exponentially large when the depth or number of templates grows. The predefined depth and a small number of templates could significantly limit the class of learned programs. Lastly, neural approaches face challenges of generalizability and data efficiency.

***Example-guided search techniques.*** Following the categorization introduced in this paper, example-guided techniques have previously been used for synthesizing regular expressions, string transformations, and spreadsheet operators. FlashFill [24], a tool available in Microsoft Excel to synthesize string transformations, uses input-output pairs to generate trace expressions that map inputs to outputs, and then uses these trace expressions to construct a program which is consistent with the examples. FlashRelate [9], a tool for extracting relational data from semi-structured spreadsheets, uses positive examples to generate a graph of constraints and then reduces the synthesis problem to computing a minimum spanning tree on this graph. Example-guided techniques have also been used for filtering spreadsheet data and synthesis of data completion scripts [59, 60].

Beyond synthesis, problems in domains such as graph search, decision tree learning, and grammatical inference also use example-guided techniques. Exact and approximate algorithms for problems such as graph labelling [12, 26], minimum Steiner tree [29], and the traveling salesperson problem [10] heuristically leverage the patterns in the input graph to optimize the search. Example-guided search also features in machine learning, in particular for decision tree learning algorithms such as ID3 and its variants. These algorithms calculate information gain corresponding to each attribute in the training data and use it to construct the tree in a bottom-up fashion [23, 45]. Algorithms for automata learning—such as for finding the smallest automaton which correctly classifies a given set of labelled examples [6, 11, 21]—also similarly exploit patterns in the training data.

## 8 Conclusion

We identified a class of synthesis techniques called Example-Guided Synthesis. The essence of this approach is to use the common patterns in the structure of input and output examples to limit the search only to programs that explain the data. We demonstrated EGS for synthesizing relational queries from input-output examples. We evaluated EGS on a diverse suite of tasks from the literature, and compared it to state-of-the-art synthesizers. EGS is able to synthesize or show unrealizability for all the tasks in a few seconds, and produces programs that generalize better.

EGS can find application in designing developer tools that allow non-expert end-users to generate relational queries, which can be challenging to write, by providing only a small set of input-output examples. Another compelling use case is writing program analyses. Information from the analyzed programs can be extracted and represented as relational data [8]. The user can provide a set of output examples (for instance by highlighting the section of code in an IDE), and EGS can synthesize a hypothesis explaining the highlighted outputs. An important challenge in these applications is in improving the sample efficiency of the learning algorithm. We emphasize Table 1 refers to the number of tuples in a single example, rather than the number of labelled databases needed. Additionally, while the problem of sample efficiency is orthogonal to our goals in this paper, we expect the use of background knowledge [58], active learning [7], and interactive feedback mechanisms [36, 46, 63] to significantly reduce the amount of data required for learning.

While EGS currently targets a rich set of features including multi-way joins, union, and negation, we intend to extend it in future to other useful features such as aggregation and recursion. We also plan to explore extensions of EGS to settings that involve larger input data, interactively labeled output data, and noise in the examples.

## Acknowledgments

We thank our shepherd, Xinyu Wang, and the anonymous reviewers, whose feedback substantially improved this paper. This research was supported in part by grants from the AFRL (#FA8750-20-2-0501), ONR (#N00014-18-1-2021), and the NSF (#1836822).

## References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of Databases: The Logical Level* (1st ed.). Pearson.

[2] 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)*.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*.

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[5] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93.

[6] Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978).

[7] Behnaz Arzani, Kevin Hsieh, and Haoxian Chen. 2021. Interpret-able feedback for AutoML systems. *arXiv preprint arXiv:2102.11267* (2021).

[8] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

[9] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. *SIGPLAN Not.* 50, 6 (June 2015), 218–228.

[10] M. Bellmore and G. L. Nemhauser. 1968. The Traveling Salesman Problem: A Survey. *Operations Research* 16, 3 (1968).

[11] A. W. Biermann and J. A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. 21, 6 (1972).

[12] Daniel Brélaz. 1979. New Methods to Color the Vertices of a Graph. *Commun. ACM* 22, 4 (April 1979), 251–256.

[13] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. 2020. Turning 30: New Ideas in Inductive Logic Programming. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*.

[14] Andrew Cropper and Stephen Muggleton. 2015. Logical Minimisation of Meta-Rules Within Meta-Interpretive Learning. In *Inductive Logic Programming*.

[15] Jacek Czerniak and Hubert Zarzycki. 2003. Application of Rough Sets in the Presumptive Diagnosis of Urinary System Diseases. In *Artificial Intelligence and Security in Computing Systems*.

[16] 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)*.

[17] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[18] Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *Journal of Artificial Intelligence Research* 61 (2018).

[19] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[20] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[21] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978).

[22] Todd J. Green. 2015. LogiQL: A Declarative Language for Enterprise Applications. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*.

[23] J. Grzymala-Busse. 1993. Selected Algorithms of Machine Learning from Examples. *Fundam. Informaticae* 18 (1993).

[24] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330.

[25] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Communications of the ACM (CACM)* 55, 8 (2012).

[26] Dorit S. Hochba. 1997. Approximation Algorithms for NP-Hard Problems. *SIGACT News* 28, 2 (1997).

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

Springer International Publishing, Cham, 335–352.

[28] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis.

[29] Marek Karpinski and Alexander Zelikovsky. 1997. New approximation algorithms for the Steiner tree problems. *Journal of Combinatorial Optimization* 1, 1 (1997).

[30] Makr Law. 2018. *Inductive Learning of Answer Set Programs*. Ph.D. Dissertation. Imperial College London.

[31] Mark Law, Alessandra Russo, and Krysia Broda. 2014. Inductive Learning of Answer Set Programs. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*.

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

[33] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[34] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[35] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[36] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[37] M. Martin, B. Livshits, and M. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[38] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[39] Stephen Muggleton. 1991. Inductive Logic Programming. *New Generation Computing* 8, 4 (Feb. 1991).

[40] Stephen Muggleton. 1995. Inverse Entailment and Progol. *New Generation Computing* 13, 3 (1995).

[41] Stephen Muggleton and Cao Feng. 1990. Efficient Induction Of Logic Programs. In *New Generation Computing*. Academic Press.

[42] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. 34, 3 (2009).

[43] Gordon D. Plotkin. 1970. A Note on Inductive Generalization. *Machine Intelligence* 5 (1970), 153–163.

[44] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[45] J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986).

[46] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 722–735.

[47] 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)*.

[48] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*.

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

[50] Stefan Schoenmackers, Oren Etzioni, Daniel S. Weld, and Jesse Davis. 2010. Learning First-Order Horn Clauses from Web Text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[51] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering queries based on example tuples. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[52] 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)*.

[53] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Advisor(s) Bodik, Rastislav.

[54] 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 (ASPLOS XII)*. ACM, 404–415.

[55] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*.

[56] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[57] 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 (PLDI 2017)*. ACM, 452–466.

[58] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *Proceedings of the 43rd International Conference on Software Engineering (To appear) (ICSE)*.

[59] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (Oct. 2017), 26 pages.

[60] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data Using Examples. *SIGPLAN Not.* 51, 10 (Oct. 2016), 195–213.

[61] 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)*.

[62] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Proceedings of the USENIX Security Symposium*.

[63] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 57 (Oct. 2017), 30 pages.

## A Runtime Comparisons

**Table 3.** Performance of EGS, Scythe, ILASP, and ProSynth on 20 knowledge discovery benchmarks.

| **Benchmark** | **EGS** | **Scythe** | **ILASP** *Task-Agnostic Rule Set* | **ILASP** *Task-Specific Rule Set* | **ProSynth** *Task-Agnostic Rule Set* | **ProSynth** *Task-Specific Rule Set* | **#Rules** *Task-Agnostic* | **#Rules** *Task-Specific* |
|---|---|---|---|---|---|---|---|---|
| *Knowledge Discovery* | | | | | | | | |
| abduce | 0.4 | – | – | 6.1 | – | – | – | 4917 |
| adjacent-to-red | 0.4 | 1.5 | 365.7 | 0.3 | – | 0.8 | 209799 | 101 |
| agent | 0.8 | – | – | 0.3 | – | 1.8 | – | 142 |
| animals | 0.4 | – | – | – | – | – | 1242184 | 2000 |
| cliquer | 0.3 | 0.7 | 1.0 | 0.2 | – | 0.7 | 1484 | 79 |
| contains | 0.3 | 0.8 | 176.1 | 0.2 | – | 0.1 | 7557 | 1 |
| grandparent | 0.5 | – | – | 5.9 | – | – | – | 4917 |
| graph-coloring | 0.4 | 5.2 | 177.2 | 0.1 | – | 0.3 | 96079 | 23 |
| headquarters | 0.3 | 0.7 | 11.2 | 0.2 | – | 0.1 | 4057 | 1 |
| inflammation | 0.6 | – | – | 3.0 | – | – | – | 847 |
| kinship | 0.5 | – | – | 5.8 | – | – | – | 4917 |
| predecessor | 0.2 | 1.7 | 1.2 | 0.2 | – | 0.1 | 1484 | 5 |
| reduce | 0.3 | 0.7 | 114.8 | 0.1 | – | 0.1 | 7557 | 1 |
| scheduling | 0.4 | 1.5 | 336.7 | 0.1 | – | 0.2 | 160016 | 16 |
| sequential | 0.8 | – | – | – | – | – | – | – |
| ship | 0.3 | 1.3 | – | 1.2 | – | – | – | 1426 |
| son | 0.3 | 1.1 | – | 1.0 | – | – | – | 1199 |
| traffic | 0.5 | 6.5 | 143.9 | 0.3 | – | 0.7 | 93326 | 97 |
| trains | 0.4 | – | – | 3.3 | – | – | – | 601 |
| undirected-edge | 0.3 | 1.0 | 1.3 | 0.2 | – | 0.3 | 1484 | 79 |

**Table 4.** Performance of EGS, Scythe, ILASP, and ProSynth on 18 program analysis benchmarks.

| **Benchmark** | **EGS** | **Scythe** | **ILASP** *Task-Agnostic Rule Set* | **ILASP** *Task-Specific Rule Set* | **Prosynth** *Task-Agnostic Rule Set* | **Prosynth** *Task-Specific Rule Set* | **#Rules** *Full* | **#Rules** *Task-Specific* |
|---|---|---|---|---|---|---|---|---|
| *Program Analysis* | | | | | | | | |
| arithmetic-error | 0.2 | 1.0 | – | 0.1 | – | 0.1 | 263853 | 13 |
| block-succ | 0.4 | – | – | 9.9 | – | – | – | 9758 |
| callsize | 0.3 | 1.2 | 20.2 | 0.2 | – | 0.4 | 14446 | 11 |
| cast-immutable | 0.3 | 1.2 | 420.0 | 0.1 | – | 0.1 | 225108 | 18 |
| downcast | 1.8 | – | – | – | – | – | – | 3392 |
| increment-float | 0.3 | 1.6 | 58.5 | 0.1 | – | 0.1 | 19594 | 10 |
| int-field | 0.3 | 0.5 | – | 0.3 | – | 0.2 | – | 109 |
| modifies-global | 0.3 | 0.7 | 23.3 | 0.1 | – | 0.1 | 17679 | 6 |
| mutual-recursion | 0.3 | 1.3 | 1.2 | 0.2 | – | 0.1 | 1484 | 25 |
| nested-loops | 2.9 | – | – | 1.1 | – | – | – | 1053 |
| overrides | 0.3 | 1.2 | – | 1.6 | – | – | – | 1804 |
| polysite | 3.8 | – | – | – | – | – | – | 1025 |
| pyfunc-mutable | 0.4 | 2.0 | 17.0 | 0.2 | – | 0.1 | 12185 | 6 |
| reach | 0.3 | 1.0 | 545.3 | 0.3 | – | 0.2 | 256549 | 15 |
| reaching-def | 0.2 | 0.8 | – | 0.3 | – | 0.1 | – | 8 |
| realloc-misuse | 0.4 | – | – | 0.1 | – | 0.2 | 669744 | 22 |
| rvcheck | 0.6 | – | – | 29.0 | – | – | – | 20186 |
| shadowed-var | 0.3 | 1.8 | – | 0.3 | – | 0.2 | 13291 | 38 |

**Table 5.** Performance of EGS, Scythe, ILASP, and ProSynth on 41 database querying tasks.

| **Benchmark** | **EGS** | **Scythe** | **ILASP** | **ILASP** | **Prosynth** | **Prosynth** | **#Rules** | **#Rules** |
|---|---|---|---|---|---|---|---|---|
| | | | *Task-Agnostic Rule Set* | *Task-Specific Rule Set* | *Task-Agnostic Rule Set* | *Task-Specific Rule Set* | *Task-Agnostic* | *Task-Specific* |
| *Relational Queries* | | | | | | | | |
| sql01 | 0.4 | 1.4 | 108.3 | 0.3 | – | 2.7 | 82475 | 200 |
| sql02 | 0.2 | 1.5 | 21.0 | 0.3 | – | 1.9 | 22073 | 212 |
| sql03 | 0.4 | 4.7 | – | 1.2 | – | 22.6 | 381295 | 752 |
| sql04 | 0.4 | 3.3 | – | 0.2 | – | 0.1 | 763408 | 2 |
| sql05 | 0.2 | 2.5 | 4.6 | 0.2 | – | 0.1 | 1571 | 1 |
| sql06 | 0.3 | 1.2 | – | 0.5 | – | 0.1 | – | 21 |
| sql07 | 0.6 | 3.7 | – | 0.2 | – | 0.3 | 258271 | 36 |
| sql08 | 0.3 | – | 21.4 | 0.1 | – | 0.1 | 14415 | 11 |
| sql09 | 0.4 | 2.1 | – | 0.4 | – | 0.1 | – | 8 |
| sql10 | 0.3 | 8.4 | 1.4 | 0.1 | 2.8 | 0.1 | 331 | 1 |
| sql11 | 0.2 | 49.7 | 40.8 | 0.2 | – | 0.1 | 14415 | 11 |
| sql12 | 0.3 | 4.1 | – | 0.2 | – | 0.1 | – | 2 |
| sql13 | 0.3 | 3.0 | – | 0.1 | – | 0.1 | 86032 | 3 |
| sql14 | 0.4 | 2.3 | – | 0.2 | – | 0.1 | 182739 | 8 |
| sql15 | 0.6 | 2.4 | – | 1.1 | – | – | – | 1461 |
| sql16 | 0.4 | 10.4 | – | 0.6 | – | 0.1 | – | 3 |
| sql17 | 0.3 | 4.8 | – | 0.2 | – | 0.1 | 187020 | 2 |
| sql18 | 0.2 | 2.1 | 31.9 | 0.1 | – | 0.1 | 14403 | 1 |
| sql19 | 0.5 | 3.1 | – | 1.7 | – | – | – | 1832 |
| sql20 | 0.2 | 1.5 | 0.8 | 0.2 | 5.5 | 0.1 | 344 | 1 |
| sql21 | 0.3 | 3.5 | 325.0 | 0.1 | – | 0.1 | 86032 | 3 |
| sql22 | 1.9 | 6.3 | 92.5 | 0.2 | – | 1.1 | 54821 | 51 |
| sql23 | 0.3 | 6.3 | – | 0.1 | – | 0.1 | 2037 | 2 |
| sql24 | 0.2 | 1.4 | 10.9 | 0.1 | – | 0.1 | 1958 | 2 |
| sql25 | 0.4 | 17.0 | 13.4 | 0.1 | – | 0.3 | 8946 | 9 |
| sql26 | 0.3 | 14.7 | 11.2 | 0.1 | – | 0.1 | 4445 | 4 |
| sql27 | 0.5 | 5.9 | 22.6 | 0.1 | – | 0.2 | 13810 | 18 |
| sql28 | 0.3 | 8.2 | 403.5 | 0.2 | – | 0.1 | 181232 | 22 |
| sql29 | 0.3 | 1.0 | – | 0.3 | – | 0.2 | – | 76 |
| sql30 | 0.3 | 3.1 | – | 0.3 | – | 0.1 | 763408 | 2 |
| sql31 | 0.4 | 2.1 | 73.1 | 0.2 | – | 2.5 | 53813 | 166 |
| sql32 | 0.3 | 17.0 | 418.3 | 0.1 | – | 0.1 | 225108 | 18 |
| sql33 | 0.4 | 2.8 | – | 4.6 | – | – | – | 6632 |
| sql34 | 0.2 | 3.2 | 540.3 | 0.1 | – | 0.1 | 225108 | 18 |
| sql35 | 0.7 | – | – | 0.5 | – | 0.1 | – | 4 |
| sql36 | 32.6 | 199.8 | – | – | – | – | – | 247986 |
| sql37 | 0.7 | 12.7 | – | – | – | – | – | – |
| sql38 | 0.5 | – | 22.0 | 0.3 | – | 0.2 | 13810 | 18 |
| sql39 | 6.8 | 11.7 | – | 4.9 | – | 1.7 | – | 325 |
| sql40 | 0.3 | 6.5 | – | – | – | – | – | 559577 |
| sql41 | 0.2 | 3.9 | – | 0.1 | – | 0.1 | – | 44 |