Syntax-Guided Synthesis of Datalog Programs

Anonymous Author(s)

ABSTRACT

Datalog is used to express recursive rule-based programs in a variety of application domains. We propose a programming-by-example system called ALPS to synthesize such programs from input-output examples. Scaling synthesis to realistic programs in this manner is challenging due to the rich expressivity of Datalog. We present a syntax-guided synthesis approach that prunes the search space by exploiting the observation that in practice Datalog programs comprise rules with similar latent syntactic structure. We evaluate ALPS on 34 benchmarks from three domains—knowledge discovery, program analysis, and database queries. The evaluation shows that ALPS can synthesize 33 of these benchmarks using a small set of examples, and outperforms state-of-the-art tools Metagol and Zaatar, which can synthesize only up to 10 of the benchmarks.

1 INTRODUCTION

Deductive rule-based programs are ubiquitous. Datalog, a declarative logic programming language, is commonly used to express such programs. It originated as a query language for deductive databases but has found applications in a variety of domains, including bioinformatics [24, 48], big-data analytics [21, 50, 53], natural language processing [33], networking [30], robotics [46], and program analysis [9, 16], among many others.

Recently, there has been a surge of interest in programming-by-example (PBE) techniques, which aim to learn a program that satisfies a specification given in the form of a set of input-output examples [2, 10, 13, 14, 18, 23]. The primary motivation underlying this interest is to simplify software development: In the case of end users, input-output examples are a simple means for demonstrating a desired computation (e.g., FlashFill [19]). In the case of advanced users, examples can aid in developing intricate algorithms by automating parts of the development process [43].

Such considerations also motivate the need to learn Datalog programs from examples. For instance, it can aid in programming network components in software-defined networking [30] or writing queries over massive graphs in big-data analytics [50, 53]. While ostensibly simple, however, learning such programs is challenging because Datalog is powerful enough to capture all polynomial time computations due to its support for recursion. Learning logic programs from examples has been extensively studied in a subfield of machine learning called inductive logic programming (ILP) [34, 38]. However, even state-of-the-art ILP techniques are very limited in their ability to learn recursive Datalog programs [3, 39].

In this paper, we propose a new approach for learning Datalog programs. Specifically, we observe that such programs in practice comprise rules that have similar latent syntactic structure. Our approach exploits this insight via the syntax-guided program synthesis paradigm [4], wherein the syntactic structure of the target class of programs is leveraged to efficiently traverse the hypothesis space of programs. For this purpose, our approach must address three key challenges: (i) capture syntactic structure effectively, (ii) minimize the number of examples needed, and (iii) explore the search space efficiently. We next elaborate upon each of these objectives.

To capture the syntactic structure of rule-based programs, we use meta-rules [39]—templates that describe a set of possible rules that can appear in a program. The key challenge is to obtain a set of meta-rules that is general enough to capture useful programs but specific enough to enable efficient synthesis. We propose a novel approach to systematically generate meta-rules, taking advantage of domain knowledge.

To minimize the number of examples needed, our approach aims to ask an oracle (e.g., a human user) a small number of queries concerning the expected output on a given input. The user need only answer with yes or no, and is not burdened with crafting meaningful examples and manually supplying them to the synthesizer. We use an active learning technique, called query by committee (QBC) [15, 52], to pick an example that can prune the search space the most. In our setting, QBC takes as input a committee formed by a set of consistent programs, and returns an example on which the committee disagrees the most—the most controversial example.

We then prune the programs that disagree with the given label and repeat the process. However, it is infeasible to apply QBC on the entire search space, which is prohibitively large in practice.

To explore the search space efficiently and overcome the challenge in using QBC, we use a bidirectional synthesis strategy to maintain the most-general and most-specific programs that are consistent with the given examples [32]. Intuitively, the most-general and most-specific programs (defined through logical entailment) form a representative set of the search space, allowing us to preserve exactness of the search. Moreover, this set is much smaller than the size of the search space, making it an ideal committee. To incrementally update the search space as examples are labeled, we define efficient top-down and bottom-up refinement operators that are guided by the given set of meta-rules.

We have implemented our techniques on top of a system called ALPS and evaluated it on a diverse set of benchmarks comprising 34 programs divided into three categories: knowledge discovery, program analysis, and database queries. The evaluation shows that ALPS can synthesize 33 of these benchmarks using a small set of examples, and outperforms state-of-the-art tools Metagol [10], Zaatar [3] and Scythe [59] which can synthesize only up to 14 of the benchmarks.

We summarize the main contributions of this paper:

- We present a syntax-guided technique for synthesizing recursive Datalog programs from input-output examples.
- Our technique employs a novel bidirectional search strategy to efficiently traverse the space of possible programs.
- Our technique minimizes the number of required examples using an active learning technique called query-by-committee.
• We demonstrate the effectiveness of our technique at synthesizing Datalog programs from diverse domains and its ability to outperform existing state-of-the-art techniques.

2 OVERVIEW EXAMPLES

In this section, we provide illustrative examples. Given an instance of input and output relations as input, our tool ALPS aims to learn Datalog programs correct with respect to the input. The desired output relations can be provided either upfront or in an interactive manner; ALPS iteratively poses membership queries of examples in the output relations. The queries are answered by an oracle, such as a human user, a dataset, or a reference implementation. Each query posed by ALPS is meant to help it disambiguate between candidate programs and converge to the oracle’s desired program.

In the learning process, it is crucial to avoid both overfitting and underfitting to the input. We demonstrate three illustrative use cases of ALPS that highlight the following aspects.

• ALPS avoids underfitting to the input by generating programs that correctly classify all examples. Previous ILP tools [35, 47, 62] are often interested in programs that correctly classify most examples.

• ALPS avoids overfitting to the input by generating multiple programs as a succinct representation of the version space, the set of all correct programs in search space. Among them, the most desirable one can be chosen for future use.

• ALPS improves the richness of programs in search space through inventing predicates that do not occur in the input.

Example 2.1 (Program analysis). Datalog has been commonly used as a language to concisely express program analyses [40, 49, 55]. However, writing a practical analyzer in Datalog is a highly non-trivial task even for experts.

We demonstrate how ALPS can be used to learn a static analysis that detects API misuse. API misuse (e.g., incorrect use of SSL API) is a well-known source of bugs that can cause serious security vulnerabilities. We can obtain a correct static API misuse detector in the following manner. For a given example program with known API misuses which can be obtained by manual inspection or using an automated tool such as [61], we populate input relations representing the syntax of the program and output relations representing the bugs. Then, ALPS learns Datalog rules which detect API misuse. Consider the following C program using the OpenSSL API. Functions ssl_socket_open1–4 establish a SSL socket and return a constant OK if they succeed. Two functions ssl_socket_open2 and ssl_socket_open3 misuse the API because they wrongly return OK when a SSL socket is not properly established.

Our goal is to learn a Datalog program that detects functions containing incorrect uses of the SSL API. The program uses the following two OpenSSL API functions.

• SSL_get_peer_certificate returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

• SSL_get_verify_result returns the result of the verification of the X509 certificate presented by the peer, if any. It returns a constant named X509_V_OK if the verification succeeded or no peer certificate was presented.

Functions should return OK only when (i) SSL_get_peer_certificate returns a non-null pointer, and (ii) SSL_get_verify_result returns a constant X509_V_OK.

```c
int ssl_socket_open1(SSL* ssl) {
    X509* cert = SSL_get_peer_certificate(ssl);
    long err = SSL_get_verify_result(ssl);
    if (!cert) {...}
    if (err == X509_V_OK {...
        return OK; // correct
    }
}

int ssl_socket_open2(SSL* ssl) {
    X509* cert = SSL_get_peer_certificate(ssl);
    if (cert == NULL) {...
     long err = SSL_get_verify_result(ssl);
    ...
    return OK; // incorrect (missing check on err)
    }
}

int ssl_socket_open3(SSL* ssl) {
    long err = SSL_get_verify_result(ssl);
    if (err != X509_V_OK {...
    X509* cert = SSL_get_peer_certificate(ssl);
    if (cert) {...
        return OK; // correct
    }
}

int ssl_socket_open4(SSL* ssl) {
    long err = SSL_get_verify_result(ssl);
    switch(err) {
        case X509_V_OK:
            cert = SSL_get_peer_certificate(ssl);
        ...
        return OK; // incorrect (missing check on cert)
    }
}
```

The problem involves four input relations and one output relation with the following meaning:

• opSucc(l1, l2): Program execution may flow from line l1 to l2.

• Certificate(x, l): The return value of function SSL_get_peer_certificate is assigned to the variable x at line l.

• Verify(x, l): The return value of function SSL_get_verify_result is assigned to the variable x at line l.

• Check(x, l): The value of variable x is checked if it is (not) equal to a specific value at line l.

• Ok(l): The function that returns OK at line l correctly uses SSL API.

We provide a relational representation of the analyzed C program, namely

```c
Certificate(2, cert), Verify(3, err), Check(cert, 4), Check(err, 5), ...
```

along with Ok(6) and Ok(12) as positive examples and the Ok(14) and Ok(31) as negative examples of the output relation to ALPS. Within 6 minutes, ALPS generates four programs, which are the most-general and most-specific programs of the version space. After manual inspection, we choose the following program as the most desirable. TODO: Should we mention something about how all four programs are equivalent?

```c
CertFlow(XL2) :- Certificate(XL1), opSucc(XL1, XL2).
```
The new predicates CertFlow(x,l), VeriFlow(x,l), and VeriCheck(l) are invented by Alps¹ to improve the richness of programs in search space. How the search space is determined will be detailed in Section 4.2. The relation CertFlow(x,l) (VeriFlow(x,l), resp.) indicates the return value of SSL_get_peer_certificate (SSL_get_verify_result, resp.) is checked if it is (not) equal to a specific value and execution flows to line l. The relation CertCheck(l) (VeriCheck(l) resp.) means the return value of SSL_get_peer_certificate (SSL_get_verify_result resp.) is the return value of

Definition 2.2 (Relational Queries). Recent work has addressed the automatic synthesis of queries over relational data [59, 63]. Datalog has been widely adopted as a language to query relational data due to its expressiveness and high-performance query evaluation engines [6, 17, 49].

By using Alps, we can obtain relational queries from input-output examples. Suppose we want to synthesize a relational query to find the students who takes different classes on the same day from a database. The problem involves four input relations and one output relation with the following meaning:

- Student(s,n): The student s is associated with ID n.
- Class(c,d): The class c is on the day d.
- Enrolled(n,c): The student with ID n is enrolled in the class c.
- Busy(s): The student s takes 2 different classes on the same day.

Given 7 positive and 7 negative examples of the output relation Busy along with input relations regarding 14 students and 6 classes, Alps synthesizes the following Datalog program within 18 seconds (in this example, only a single program is generated since the provided examples ensure unambiguity).

```
EnrollClass(n,c,l) :- Enrolled(n,c), Class(c,l).
Busy(s) :- Student(s,n), EnrollClass(n,c1,l), EnrollClass(n,c2,l), c1 /= c2.
```

where EnrollClass is an invented predicate. While ostensibly simple, the above query is non-trivial to be synthesized since it is semantically equivalent to the following complex SQL query:

```
SELECT s,n FROM Student S
WHERE s,n IN (SELECT E1.n
  FROM Enrolled E1, Enrolled E2, Class C1, Class C2
  WHERE E1.n = E2.n AND E1.c = <E2.c
  AND E1.c = C1.c AND E2.c = C2.c AND C1.d = C2.d)
```

The state-of-the-art tool Scythe [59] for synthesizing SQL queries fails to generate the above SQL query within 3 hours.

Definition 2.3 (Knowledge Discovery). Knowledge discovery aims to extract useful knowledge from databases. We test the feasibility of using Alps to discover interesting correlations between patient risk factors and a disease called acute inflammations of urinary bladder that may warrant further investigation. We used a dataset created by a medical expert to enable expert systems that perform presumptive diagnosis of the disease [11]². The dataset comprises information of 120 patients. The problem involves 6 input relations and one output relation with the following meaning:

- Temperature(p,t): The temperature of the patient p is t degree centigrade.
- Nausea(p): The patient p is with nausea.
- Lumbar_pain(p): The patient p is with lumbar pain.
- Urine_pushing(p): The patient p is with continuous need for urination.
- Micturition_pushing(p): The patient p is with micturition pain.
- Burning_urethra(p): The patient p is with burning of urethra.
- Inflammation(p): The patient p may be diagnosed with acute inflammation of urinary bladder.

To avoid overfitting to the dataset, we set aside information of randomly chosen 20 patients for validation and use information of the other 100 patients for training. Alps synthesizes the following four programs within 4 seconds.

```
// Program 1
Inflammation(p) :- not Lumbar_pain(p), Urine_pushing(p).
Inflammation(p) :- Nausea(p), Urine_pushing(p).

// Program 2
Inflammation(p) :- not Lumbar_pain(p), Urine_pushing(p).
Inflammation(p) :- Micturition_pushing(p), Urine_pushing(p).

// Program 3
Inflammation(p) :- not Burning_urethra(p), Urine_pushing(p).
Inflammation(p) :- not Burning_urethra(p), Micturition_pushing(p).

// Program 4
Inflammation(p) :- not Burning_urethra(p), Urine_pushing(p).
Inflammation(p) :- Micturition_pushing(p), Urine_pushing(p).
```

For each program, we compute the F-score to evaluate the prediction accuracy on the validation set. The calculated F-scores are 1.0, 1.0, 0.7 and 0.9, respectively. We can pick either one of the first two programs because they show the best predictive performance.

Note that other learning algorithms (such as decision trees or SVMs) do not generate we cannot obtain the most desirable classifier in the entire search space. This is because those algorithms do not contain all correct classifiers in their hypothesis space.

3 PROBLEM FORMULATION

In this section, we present key concepts and formalize the synthesis problem.

3.1 Datalog Programs

Rules. A term t is either a variable x, y, z, . . . , or a constant a, b, c, . . . . A relation symbol p, q, r, . . . is associated with an arity ar(r). An atom is an application of a relation symbol to a vector of variables and constants, e.g., t(x,y,a) for a relation r with arity 3. A ground atom is an application of a relation symbol to constants, e.g., r(a1, . . . , ar). A Datalog rule C is an expression of the form:

```
A :- B1, B2, . . . , Bn.
```

¹For the reader’s convenience, we changed the relation names to reveal their meanings (Alps uses mechanically generated names for invented predicates).

²Available at http://archive.ics.uci.edu/ml/datasets/Acute+Inflammations.
where \( A, B_1, \ldots, B_n \) are atoms. The atom \( A \) is called the head of the rule; the set of atoms \( \{B_1, \ldots, B_n\} \) is called the body of the rule. A Datalog rule can be interpreted as a logical implication: if \( B_1, \ldots, B_n \) are true, then so is \( A \). In this paper, we consider only positive rules, which do not contain negation in their bodies.

**Programs.** A Datalog program \( P \) is a finite set of rules. We divide relation symbols into two categories: the input relations whose contents are given, and the output relations whose contents are derived from the input relations using the program \( P \). An input relation can never appear in the head of a rule. We use \( I \) to denote the set of facts (ground atoms) in the input relations. The Herbrand base \( \mathcal{B} \) denotes all possible applications of the output relations to vectors of constants in \( I \). A Datalog program is recursive if a relation symbol appears in both the head and the body of a rule.

Semantically, evaluating \( P \) on \( I \) yields a minimal Herbrand model of \( P \cup I \), which is the smallest set of ground atoms that satisfies the rules in \( P \) and \( I \). Given a ground atom \( e \), \( P \cup I \models e \) denotes that \( P \) with input \( I \) derives fact \( e \).

### 3.2 Synthesis Problem

Our task is to learn (synthesize) Datalog programs through examples. An example here is a ground atom from the Herbrand base \( \mathcal{B} \), which can be labeled as positive (+) or negative (−). We are now ready to define our synthesis problem.

**Definition 3.1 (Synthesis problem).** A synthesis problem \( S \) is a tuple \((\mathcal{H}, O, I)\), where

- \( \mathcal{H} \) is a set of Datalog programs, i.e. the hypothesis space;
- \( O \) is an oracle that labels each example with \((+,−)\);
- \( I \) is a set of inputs—facts in the input relations.

Let \( E^+ \) be \( \{ e \in \mathcal{B} \mid O(e) = + \} \) and \( E^- \) be \( \{ e \in \mathcal{B} \mid O(e) = − \} \) be the positive and negative examples defined by the oracle respectively. The goal is to find \( P \in \mathcal{H} \) such that: for all \( e \in E^+ \), \( P \cup I \models e \), and for all \( e \in E^− \), \( P \cup I \nmodels e \).

Given a synthesis problem \((\mathcal{H}, O, I)\), the set of all Datalog programs \( P \in \mathcal{H} \) that are consistent with \( E = (E^+, E^-) \) is called the version space, and is denoted \( \mathcal{V}_E \).

### 4 OUR APPROACH

In this section, we formalize our synthesis algorithm, ALPS. In Section 4.1, we describe a running example which will be used throughout this section. In Section 4.2, we describe the structure of the search space. Section 4.3 describes the algorithm parameterized by refinement operators. Section 4.4 presents an instantiation of the algorithm with meta-rules. Section 4.5 shows subtleties involved in our choice of meta-rules and how we address them. Finally, Section 4.6 states formal properties of our algorithm.

#### 4.1 Running Example

**Example 4.1 (Graph transitive closure).** We demonstrate how ALPS synthesizes a graph query: computing the transitive closure of a directed graph. The problem involves one input relation \( \text{edge} \) and one output relation \( \text{path} \) with the following meaning:

- \( \text{edge}(x, y) \): there is an edge from node \( x \) to node \( y \).
- \( \text{path}(x, y) \): there is a path from node \( x \) to node \( y \).

Suppose the user populates the input relation \( \text{edge} \), with the following example graph:

![Example Graph](image)

where an edge from node \( i \) to node \( j \) indicates that \( \text{edge}(i, j) \) appears in the input relation. Note that it is natural for users to provide such an input to illustrate what they want to achieve. ALPS then iteratively asks queries about the contents of the output relation. In this example, it begins by posing a membership query, e.g.: Is there a path from 1 to 2? It obtains the answer yes. Using this information, ALPS poses the next question: Is there a path from 3 to 2? It obtains the answer no, and the process continues. In each step of the process, ALPS seeks to prune the space of candidate programs—the version space—with the goal of arriving at an unambiguous solution. In this example, after 4 queries (out of 49 possible queries—all pairs \((i, j)\) where \( i, j \in \{1, 2, 3, 4, 5, 6, 7\} \), ALPS arrives at the following recursive program, which computes the transitive closure of a directed graph.

\[
\begin{align*}
\text{path}(x, y) & \leftarrow \text{edge}(x, y). \\
\text{path}(x, z), \text{path}(z, y) & \leftarrow \text{path}(x, y).
\end{align*}
\]

In fact, since ALPS maintains the version space of consistent programs, it also discovers the following non-linear recursive program:

\[
\begin{align*}
\text{path}(x, y) & \leftarrow \text{edge}(x, y). \\
\text{path}(x, z) & \leftarrow \text{path}(x, y), \text{path}(y, z).
\end{align*}
\]

Note that both of the recursive rules above are very similar: the only difference is in the relation names. We can formalize this by considering meta-rules [39], which are essentially Horn clauses where the relation names are kept abstract and can be instantiated later. Then both of the recursive rules are instances of the following meta-rule:

\[
\text{R}_{ijk}(x, z) \leftarrow \text{R}_3(x, y), \text{R}_2(y, z).
\]

In practice, many Datalog rules follow the chain pattern. This suggests a strategy for synthesizing Datalog programs: enumerate all possible instantiations of the above meta-rule with concrete relations and examine their combinations. Unfortunately, this single template is insufficient to synthesize Datalog programs, as we shall see in the following example.

#### 4.2 Structure of the Search Space

The hypothesis space \( \mathcal{H} \) consists of a finite set of Datalog programs over the same input and output relations. We will use transitive closure (Example 4.1) as the running example. To keep the presentation simple, we consider a hypothesis space \( \mathcal{H} \) where all programs use a subset of the following four rules:

\[
\begin{align*}
r_1 & : \text{path}(x, y) \leftarrow \text{edge}(x, y). \\
r_2 & : \text{path}(x, z) \leftarrow \text{path}(y, z). \\
r_3 & : \text{path}(x, x) \leftarrow \text{edge}(x, x). \\
r_4 & : \text{path}(x, y) \leftarrow \text{path}(x, z), \text{path}(z, y).
\end{align*}
\]

We will denote the Datalog program consisting of rules \( r_i, r_j, r_k \) as \( P_{ijk} \).
4.3 The ALPS Algorithm

Given a synthesis problem $S = (H, O, I)$, ALPS applies Algorithm 1 to find a solution for $S$.

The algorithm is a fixpoint algorithm: it maintains a pair $E = (E^+, E^-)$ of positive and negative examples, and a set of most-general programs $\overline{P}$ and most-specific programs $P$ that are always consistent with $E$. The examples are initially empty, and $\overline{P}, P$ are initialized to be the most general and most specific programs respectively (we define this initialization in Section 4.4). At every iteration, it adds a (positive or negative) example by querying the oracle $O$. Then, it invokes two refinement operators $F^+, F^-$ which recalculate the most-general programs and the most-specific programs that agree with the new example (we define the refinement operators in Section 4.4). The algorithm stops when no new examples can be added.

The crux of the algorithm is the way we choose the example to query the oracle. The union of two sets of programs $\overline{P}, P$ forms the committee $P$. The committee then picks the most controversial example $e^*$. If $O(e^*) = +$, then $e^*$ is added to $E^+$; otherwise, $e^*$ is added to $E^-$. If no controversial example exists, then everyone in the committee agrees; the algorithm terminates and returns the set $P$, which contains all the most-general and most-specific solutions.

In order to determine the most controversial example, we use the metric of vote entropy. It is inspired by query-by-committee [15, 52], a strategy used in active learning [51]. Since there are only two possible labels for an example, we use a simplified definition:

**Definition 4.2 (Vote entropy).** For an example $e$ and set of committee members $K$, the normalized vote entropy is:

$$D(e, K) = 1 - \frac{2}{|K|} \left| P - \frac{|K|}{2} \right|$$

where $p$ is the number of committee members that assign a positive label to the example $e$.

When the vote entropy of an example is zero, all programs in the committee agree on its label. Figure 1 shows the version space of programs and the query posed to oracle at each iteration for our running example.

### 4.4 Refinement with Meta-Rules

We now give the concrete definitions for initialization functions and refinement operators, $F^+$ and $F^-$ in Algorithm 1. The design of the refinement operators is motivated by a practical insight: the synthesis search should be biased towards patterns that are frequently used in practice. We are inspired by meta-rules [39], which are templates that dictate syntactic restrictions on rules and therefore a natural representation to bias the search.

**Meta-rules.** A meta-rule is a second-order rule. Multiple rules can be instantiated from a meta-rule. We shall use $V_1$ and $V_2$ to denote first- and second-order variables, respectively. A meta-rule takes the following form:

$$R_1(x_1, \ldots, x_m) \rightarrow R_2(y_1, \ldots, y_n), \ldots, R_k(z_1, \ldots, z_m)$$

where $x_1, y_2, z_1 \in V_1$ and $R_k \in V_2$.

A meta-rule can be instantiated by substituting second-order variables with relation symbols. For example, the rules from the running example are generated by the following meta-rules:
Algorithm 2 Meta-rule-guided refinement

Function $F^1(P, E^+, E^-)$

1: $E ← (E^+, E^-)$
2: while $P ⊈ V_E$ do
3:   $ΔP ← (P ∩ V_E) \setminus V_E$
4:   $ΔP ← P^1(ΔP, T) \setminus V_E$
5:   $P ← (P ∩ V_E) ∪ ΔP$
6: end while
7: return $P$

Function $F^1(P, E^+, E^-)$

1: $E ← (E^+, E^-)$
2: while $P ⊈ V_E$ do
3:   $ΔP ← (P ∩ V_E) \setminus V_E$
4:   $ΔP ← P^1(ΔP, T) \setminus V_E$
5:   $P ← (P ∩ V_E) ∪ ΔP$
6: end while
7: return $P$

Similar to rules, a generality order between meta-rules can be established using $θ$-subsumption by allowing substitution for second-order variables as well as first-order variables. Using this generality order, a set of meta-rules forms a partially ordered set.

Initialization. The initialization function MostGeneral() collects all rules instantiated from the most general meta-rules and combines them as the most general program. The initialization function MostSpecific() makes each individual rule instantiated from the most specific meta-rules as a single rule program, and all of these programs form the initial set of most specific programs.

Meta-rule-guided refinement. Algorithm 2 describes our refinement operations, $F^1$ and $F^1$, which are parameterized by a set of meta-rules T. We explain only top-down refinement $F^1$ in detail, since bottom-up refinement $F^1$ works in a symmetrical way.

The algorithm begins with the given set of programs $P$. Then, it iteratively specializes the programs by applying the specialization operator $P^1$, which is guided by $T$ (line 2–5). In each iteration, the condition $P \subsetneq V_E$ checks whether the current programs are consistent with the examples. If there is no violation, the algorithm terminates. Otherwise, line 3 first eliminates programs violating positive examples, and then selects programs violating negative examples to specialize. In the former case, programs fail to derive a positive example, and more specific programs will also fail to derive it. This process removes not only inconsistent programs but also any programs more specific than them. The elimination happens in the third iteration of our running example shown in Figure 1c: when $P_{23}$ is eliminated due to the positive example path$(1,2)$, all the more specific programs $P_{34}, P_{23}, P_{3}, P_{4}$ are eliminated from consideration as well.

Next, line 4 specializes programs violating negative examples by calling $P^1$, and eliminates any generated programs that fail to derive a positive example. Finally, line 5 updates $P$ by including the new specialized programs.

The final piece of the puzzle is the specialization operator $P^1$. Here, $P^1$ can specialize a program in two ways: (1) replace a rule with a more specific one; for instance, in our running example shown in Figure 1b), program $P_{12}$ is specialized to $P_{14}$ and $P_{23}$; (2) remove a rule that cannot be further specialized; for instance, $P_{23}$ could potentially be specialized to $P_{2}$. Finding all more specific rules for a given rule $r$ can be efficiently done by consulting the generality order of the meta-rules $T$: first, find the meta-rule $T_r$ used to instantiate $r$; then, find all more specific meta-rules $T_s$ with respect to $T_r$; finally examine all rules instantiated from a meta-rule in $T_s$ and keep the ones more specific than $r$.

4.5 Augmentation and Predicate Invention

The choice of meta-rules is critical for the synthesis process. If the set of meta-rules is too large, then ALPS will not be able to scale, since the search space will be huge. On the other hand, the meta-rules must be sufficiently rich to capture the desired program. Simply reusing meta-rules that are either provided by the end-user or mined from existing code repositories is usually insufficient. To solve this problem, we start with a very small set of meta-rules that we can specify manually (in our experiments we use only 3 such meta-rules), and then extend these using augmentation, a process that slightly modifies each meta-rule.

An augmentation $T'$ of a meta-rule $T$ is a meta-rule where each atom $R(x_1, \ldots, x_k)$ in $T$ is replaced by another atom $R(y_1, \ldots, y_k)$. However, we must take care to limit how much the sequence of variables changes. Denote by $d_R(T, T')$ the edit distance between the strings $x_1 \ldots x_k$ and $y_1 \ldots y_k$. Then, the augmentation distance
between \( T, T' \) is defined as
\[
AD(T, T') = \sum_R d_R(T, T')
\]
where \( R \) ranges over all atoms in \( T \). Our key idea is to consider all the augmentations of \( T \) that are within a bounded augmentation distance from \( T \). The smaller this bound is, the fewer meta-rules will be generated from \( T \). In our experiments, we could generate almost all of the programs using an augmentation distance of 5.

As an example of augmentation, consider the following 2 meta-rules:
\[
\begin{align*}
T_1 & : R_0(y) \leftarrow R_1(z), R_2(y, z). \\
T_2 & : R_0(y, z) \leftarrow R_1(z, x), R_2(y, z).
\end{align*}
\]
Then, \( T_2 \) is an augmentation of \( T_1 \) with distance 2.

The definition of augmentation distance can be naturally extended to distance between a program \( P \) and a set of meta-rules \( T \):
\[
AD(P, T) = \max_{T_1 \in T} \min_{T_2 \in T} AD(T_1, T_2)
\]
where \( T_1 \) ranges over all meta-rules that can be instantiated to a rule in \( P \).

**Predicate invention.** Another orthogonal way of improving the richness of programs in search space is predicate invention. Predicate invention helps to break a long (and complex) rule into short ones, enabling better use of simple and short templates. More importantly, it becomes unavoidable for Datalog programs with recursion. For instance, consider the example of deriving strongly connected components (SCC) from edges:
\[
\begin{align*}
\text{path}(x, y) & : \text{edge}(x, y). \\
\text{path}(x, y) & : \text{path}(x, y), \text{edge}(y, z). \\
\text{sc}(x, y) & : \text{path}(x, y), \text{path}(y, x).
\end{align*}
\]
Here, the input and output relations are \text{edge} and \text{sc}, respectively. Given that \text{sc} cannot be derived as a set of clauses in terms of the input relation \text{edge}, a new predicate like \text{path must be} invented. The difficulty with predicate invention is determining what form the invented predicates should take. Without meta-rules this becomes very difficult, because we cannot query the user about whether a tuple appears in an invented predicate and therefore we have no way of effectively constraining the syntax. With meta-rules, we can easily support predicate invention: the rules that define the potential invented predicates are exactly the instantiations of meta-rules with concrete relations.

### 4.6 Properties of ALPS

Algorithm 1 always makes progress: after every query to the oracle \( O \), we remove from consideration a controversial example from \( B \). Since the set of possible examples \( B \) is finite, the algorithm always terminates in finitely many steps. Our algorithm also guarantees that a solution is found if there are no controversial examples left in the committee. To ensure this property, it is critical that the algorithm tracks both the most-general and most-specific programs at every iteration. The following theorem succinctly captures these properties.

**Theorem 4.3.** Let \( S = (\mathcal{H}, O, I) \) be a synthesis problem such that a solution to \( S \) exists in the hypothesis space \( \mathcal{H} \). Let \( P \) be the output of ALPS. Then:

1. (Soundness) Every \( P \in \mathcal{P} \) is a solution to \( S \).
2. (Completeness) For every solution \( P \in \mathcal{H} \) to \( S \), there exist programs \( P_1, P_2 \in \mathcal{P} \) such that \( P_1 \subseteq P \subseteq P_2 \). An immediate corollary is that if there exists a program \( P \) that is a solution to \( S \), then \( P \) is nonempty.
3. (Termination) ALPS terminates in finitely many steps.

### 5 EMPIRICAL EVALUATION

We evaluate ALPS on synthesis tasks from various domains. Our implementation comprises about 8,000 lines of C++ code. It uses the fixpoint engine of the Z3 solver [22] for Datalog evaluation. The evaluation was performed on a Linux machine with 16 GB of RAM and a 3.0 GHz processor.

We designed our evaluation to investigate the following key questions about the suitability of ALPS in several settings:

1. **Q1.** How efficient is ALPS, in terms of runtime and number of queries, at synthesizing a Datalog program?
2. **Q2.** How much does meta-rule augmentation speed up the synthesis process?
3. **Q3.** How effective is the QBC selection strategy at reducing the number of queries asked?
4. **Q4.** How robust is ALPS to changes in the input data?
5. **Q5.** How does the performance of ALPS compare to similar tools for synthesis?

#### 5.1 Benchmark Suite

We use a diverse set of benchmarks comprising 34 programs divided into three main categories: knowledge discovery, program analysis and SQL, as described in Table 1. The last three columns of the table show (i) the number of input–output relations, (ii) the minimum number of rules required to synthesize the program, and (iii) whether recursion is required or not.

**Knowledge discovery.** The first category of our benchmarks is a collection of declarative programs frequently used in the artificial intelligence and database literature. We introduced the inflation benchmark in Example 2.3. The next four knowledge-discovery benchmarks (abduce, ancestor, animals, and buildWall) are widely used in the field of inductive logic programming [35, 39]. The samegen benchmark is a standard Datalog program in the database literature [1]. The path and scc benchmarks are well-known declarative programs that compute reachability and strongly connected components in a graph.

**Program analysis.** The second category consists of benchmarks from program analysis, another popular application area for declarative programming. Our benchmark suite includes 11 program-analysis tasks:

- **rv-check** is a static checker from 	exttt{apisan} [61] for finding API misuse bugs in C/C++ programs. It identifies such bugs by detecting inconsistent uses of return values of API functions. ALPS could be useful in synthesizing such rule-based tools, as incorrect rules in such tools result in large numbers of false bug reports that are cumbersome to inspect; the fact that labeling bug reports is
We observe that ALPS fails to synthesize five context sensitive pointer analysis benchmarks, as the necessary augmentation distance is too high and offers no filtering of the search space. In these cases, we include the meta-rules extracted from the remaining four benchmarks along with the chain rule. This enables ALPS to successfully synthesize four of them, as the necessary augmentation distance greatly decreases.

In addition, for each output relation of each benchmark, we upper bound the number of rules where the output relation appears in the head to 4, and upper bound the number of invented predicates to 3.

For each benchmark, ALPS starts with full knowledge of the input relations and no knowledge of output relations—i.e., it starts with no labeled examples. The size of the input relations ranges from 3 to 77, with the average size being 22. ALPS iteratively queries the oracle for the label of an example from one particular output relation. For evaluation, the oracle is an expert-written reference implementation. We manually inspected all synthesized programs to ensure that they are equivalent to the desired program.

**5.3 Evaluation Results**

**Q1: Queries and time.** Table 2 summarizes the main results of our evaluation. Consider, for instance, the ancestor benchmark. ALPS makes 11 queries to the oracle (out of a maximum of 450 queries, which is the size of the Herbrand base); it requires 25 seconds to synthesize three programs; and, in the process, it evaluates 24,280 programs out of $10^{10}$ programs in the search space.

Overall, our results demonstrate the small number of queries needed to discover non-trivial programs. For most benchmarks, we require less than 20 queries; for our largest benchmark, modref, we make 22 queries to the oracle in order to synthesize 10 rules. We synthesize most programs within a few minutes. In certain examples—like modref—a large number of programs are evaluated, thus requiring more synthesis time, which amounts to a few hours.

**Q2: Effectiveness of meta-rule augmentations.** Figure 3 shows the frequency distribution of benchmarks according to their augmentation distance with respect to chain meta-rules. As we can see, only two benchmarks can be synthesized with no augmentation, while most benchmarks—28 out of 33—can be synthesized with no more than 5 augmentations. This indicates that simple chain meta-rules are quite limited by themselves, but we can handle a large number of benchmarks by slightly mutating them.

However, with only chain meta-rules, ALPS fails to scale on five context sensitive pointer analysis benchmarks, whose augmentation distance is 6 or larger. We observe that although these five benchmarks are different from each other, their rules are quite similar. Table 3 shows the augmentation distances for each of these five benchmarks with respect to two sets of meta-rules: chain meta-rules and meta-rules mined from the other four benchmarks. The augmentation distance implies that, in comparison with general chain meta-rules, meta-rules mined from the same domain are more useful to guide the synthesis search process. Indeed, with extra meta-rules mined from the other four benchmarks, ALPS is able
to synthesize four out of five context sensitive program analysis benchmarks, as shown\(^1\) in Table 2.

\(^1\)\(\text{obj-type}\) takes ALPS 17 hours to finish, thus is marked as timeout.

**Q3: Quality of qbc.** We now investigate the quality of qbc’s selection strategy. To do so, we instrument ALPS to randomly pick an example that the committee disagrees on, instead of one that

\[\text{Figure 2: (a) Box plot of the number of queries asked by random selection (green dots mark the number of queries by qbc); (b) Number of queries asked by ALPS for the andersen benchmark under different sizes of input data (where X=7).}\]
Metagol is an ILP tool that is an instantiation of the meta-interpretive learning framework [39], which is also parameterized by meta-rules. We run Metagol with two settings: the ALPS setting, which uses the same set of meta-rules that ALPS uses after it performs augmentation, and the ideal setting, which consists of the minimal set of meta-rules that are sufficient for synthesizing a correct program. Using ALPS’s setting, Metagol cannot finish most knowledge discovery benchmarks and all program analysis benchmarks. Using the ideal setting, Metagol still fails on two knowledge discovery benchmarks and most of program analysis benchmarks. Metagol also failed on four of the SQL benchmarks despite their lack of recursion. It is important to note that Metagol employs meta-interpretive learning, which is not a complete technique, so it is not guaranteed to terminate, despite finiteness of the search space.

Zaatar [3] is a constraint-based Datalog program synthesis tool. It fails on most of our benchmarks because it is very sensitive to the size of the input data, since the size of the encoding is polynomial in the input data. In contrast, ALPS has much better scalability in terms of input size, as ALPS only evaluates candidate programs on input data instead of encoding the input as symbolic constraints.

Scythe [59] is a state-of-the-art SQL synthesis tool. SQL does not support recursion, so we only run it on the SQL benchmarks, which come with the tool. It synthesizes almost all SQL benchmarks within one minute, but fails on one benchmark due to scalability, while ALPS finishes on all of them in two minutes. This highlights our observation that by restricting the syntactic structure, e.g. chain rules, the scalability can be greatly improved.

Summary. To summarize, our experimental evaluation demonstrates (i) the small number of queries and required by ALPS to synthesize sophisticated algorithms; (ii) the significance of meta-rules and augmentations; (iii) the importance of the QBC at reducing the number of queries; and (iv) the robustness of our synthesis approach to input size.

5.4 Threats to Validity

There are several threats to the validity of our studies. The main threat to validity arises when a synthesized program does not match the user’s intentions because input relations do not cover all corner cases. We can mitigate this threat by allowing the user to provide a large input by taking advantage of ALPS’s ability to handle sizeable input data as shown in Figure (2b). In practice, large input relations often cover the vast majority of corner cases. Another threat arises when labeled examples are noisy. We can mitigate this threat by collecting answers from multiple oracles (e.g., through crowdsourcing or multiple reference implementations) and using the majority vote as the final answer. The last threat arises when a desirable program does not have common latent syntactic structure. In that case, meta-rules generated in the way we proposed may not be general enough to capture the program. We can mitigate this threat by using a larger number of invented predicates. The more invented predicates we use, the simpler rules we may obtain. Eventually, the desirable program will comprise typical rules, so that our method can be used.

Table 3: Augmentation distances of context sensitive pointer analysis benchmarks with respect to chain meta-rules and meta-rules mined from the same domain.

<table>
<thead>
<tr>
<th>Augmentation Distance</th>
<th>1-call-site</th>
<th>2-call-site</th>
<th>1-object</th>
<th>1-type</th>
<th>1-obj-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>same domain</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3: Augmentation distance distribution.
6 RELATED WORK

Inductive logic programming. While we use key ideas from inductive logic programming (ILP) in ALPS, a number of properties distinguish our approach from existing ILP approaches. First, work in ILP usually learn relations, often probabilistic ones [12], from vast amounts of mined data, e.g., biological data [36]. In our work, and in a large class of synthesis techniques, the goal is to interactively infer a program from a small, representative set of examples. Second, most ILP systems are not adept at learning recursive rules. In contrast, we specifically aim to infer recursive rules. Third, ILP is often interested in programs that correctly classify most examples. In contrast, we are interested in programs that correctly characterize all positive and negative examples. Fourth, many ILP systems require a complicated interaction model (for example, Fill [7] poses existential queries and CIGOL [37] poses generalization queries). In contrast, ALPS has a simple interaction model that only poses membership queries. Lastly, we employ a complete search strategy, whereas ILP systems can fail to find a program even if one exists that is consistent with the given examples.

More recently, ILP has been applied to end-user programming and online tutoring [20]. Meta-Interpretive Learning has been used to learn Prolog programs for string manipulation tasks [29]. These applications share similar goals as ours of learning programs by obtaining examples from users. We focus on Datalog programs whereas they learn programs in other domains, e.g., string manipulation or table transformation. Also, when their generated program is incorrect, the user is expected to provide a counterexample. In contrast, our approach automatically identifies the most controversial example to pose to the user.

Template-guided synthesis. Templates are commonly used to guide the search in program synthesis [10, 54, 57, 58]. At a high-level, meta-rules can also be seen as program sketches [57], where the holes are the relation symbols.

Interactive synthesis by example. Some programming-by-example (PBE) approaches interactively query an oracle for examples, Jha et al. [23] present an oracle-guided synthesis procedure for straight-line programs encodable in SMT. They require the oracle (usually a reference implementation) to provide the output when given some input. Another recent interactive synthesis approach is applied in the context of parser synthesis [28] to learn a grammar.

Synthesis of recursive programs. A number of works have targeted the problem of synthesizing recursive programs [2, 14, 25, 26, 42, 45]. Most of these works focus on recursive functional programs that manipulate recursive data structures. Datalog programs recursively traverse relations (hypergraphs). To our knowledge, none of the functional techniques have been applied to this domain.

Version-space algebras. Version-space algebras were used for synthesis, initially by Lau et al. [27], and more recently in FlashFill [18] for spreadsheet manipulation. ALPS maintains a version space using most-general and most-specific programs, as first proposed by Mitchell [32].

Learning for program analysis. Recently, several systems have been created that apply machine learning techniques to program analysis. Oh et al. [41] use Bayesian optimization techniques to effectively learn adaptation strategies for parametric program analysis, while Bielik et al. [8] apply the ID3 algorithm to learn a decision tree that represents points-to and allocation site facts for individual JavaScript functions. This work either learns an efficient configuration for program analysis or an accurate representation for the results of an analysis. In contrast, our work can learn the rules of the program analysis, expressed in Datalog.

7 CONCLUSIONS

We studied the problem of synthesizing Datalog programs from examples. Motivated by a variety of applications, we proposed an interactive methodology in which the user is only asked to label examples as positive or negative. We showed how to apply the active learning technique of query-by-committee that aims to minimize the number of queries by choosing the most controversial example among programs in the version space. Since it is impractical to consider all such programs, we presented a bidirectional search strategy to compute a representative set of the version space comprising only the most-general and most-specific programs. We also proposed meta-rule driven refinement operators to efficiently traverse this space as examples are labeled. We demonstrated that our approach synthesizes the desired programs for several benchmarks from diverse domains while effectively reducing the number of examples to be labeled.

REFERENCES

Anon.