Syntax-Guided Synthesis of Datalog Programs

Xujie Si*, Woosuk Lee‡, Richard Zhang*, Aws Albarghouthi‡, Paraschos Koutris‡, Mayur Naik‡

{xsi,woosuk,rmzhang,mhnaik}@cis.upenn.edu  {aws,paris}@cs.wisc.edu
1University of Pennsylvania, USA  ‡Hanyang University, South Korea  †University of Wisconsin-Madison, USA

ABSTRACT
Datalog has witnessed promising applications in a variety of domains. We propose a programming-by-example system, alps, to synthesize Datalog 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 that have similar latent syntactic structure. We evaluate alps can synthesize 33 of these benchmarks, and outperforms the state-of-the-art tools Metagol and Zaatar, which can synthesize only up to 10 of the benchmarks.

CCS CONCEPTS
• Theory of computation → Program analysis; Active learning; • Software and its engineering → Programming by example; Domain specific languages; • Information systems → Relational database query languages;

KEYWORDS
Syntax-guided synthesis, Datalog, Active learning, Template augmentation, Program analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Datalog, a declarative logic programming language, has witnessed promising applications in a variety of domains, including bioinformatics [33, 58], big-data analytics [30, 60, 63], natural language processing [44], networking [39], program analysis [17, 26], and robotics [56]. A key reason is the emergence of scalable Datalog solvers, including open-source [1, 3, 11, 59, 60] and commercial ones [2, 4, 6, 25]. Moreover, the concise and declarative nature of Datalog has made it the target of a growing body of meta-reasoning tools. For instance, program analyses written in Datalog are readily extensible with features such as fixed point frameworks [14, 40], abstraction refinement [74], and user interaction [41, 57, 73]. Likewise, software-defined networking (SDN) applications written in Datalog can avail of efficient provenance tracking to help in tasks such as debugging and repairing [71].

A key hindrance to bringing these benefits to a broad user base is the lack of automated tools to help develop Datalog programs. To this end, we propose a programming-by-example system, alps, to synthesize Datalog programs from input-output examples. This constitutes a natural next step in many domains such as program analysis and networking, where reference imperative implementations can provide input-output examples.

While ostensibly simple, synthesizing such programs is challenging because Datalog is powerful enough to capture all polynomial time computations. Learning logic programs from examples has been extensively studied in a subfield of machine learning called inductive logic programming (ILP) [45, 49]. However, even state-of-the-art ILP techniques are very limited in their ability to learn realistic Datalog programs [9, 50].

We propose a new approach to synthesize Datalog programs. Our key insight is 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 [10], 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 [50]—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 a small number of queries concerning the expected output on a given input. The oracle need only answer with yes or no, rather than crafting elaborate examples and supplying them to the synthesizer. We use an active learning technique, called query by committee (QBC) [24, 62], 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.

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 [43]. 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 end-to-end approach in the ALPS system and report on our experience evaluating it on a diverse set of 34 benchmarks from three domains: knowledge discovery, program analysis, and relational queries. The evaluation shows that ALPS can synthesize 33 of these benchmarks, and outperforms the state-of-the-art tools Metagol [19] and Zaatar [9], which can synthesize only up to 10 of the benchmarks.

We summarize the main contributions of this paper:

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

2 OVERVIEW EXAMPLES

ALPS learns Datalog programs that are correct with respect to a given instance of input and output relations. In this section, we first present two illustrative examples that highlight applications of ALPS in two domains: program analysis and relational queries. We then present an example to elucidate key design choices in ALPS. We also use it as the running example in the rest of the paper.

Example 2.1 (Program analysis). Datalog has shown great potential in the domain of program analysis [22, 40, 51, 59, 65]. Thus, there is a growing need to help synthesize program analyzers in Datalog for a variety of programming languages, including general-purpose and domain-specific ones.

We demonstrate how ALPS can be used to learn a static analysis to detect API misuses—a common source of bugs in today’s world of complex and evolving APIs. For a given example program with known API misuses, we populate input relations representing the syntax of the program and output relations representing the bugs. Then, ALPS learns Datalog rules that can be used for detecting similar API misuses.

Consider the following C program using the OpenSSL API. Functions ssl_socket_open1-4 establish a SSL socket and return a constant 0K if they succeed. Two functions ssl_socket_open2(2, 4) contain API misuses in that they incorrectly return 0K when a SSL socket is not properly established.

```c
1
2 int ssl_socket_open1(SSL* ssl)
3 X509* cert = SSL_get-peer_certificate(ssl);
4 long err = SSL_get_verify_result(ssl);
5 if (cert == NULL) {...}
6 return OK; // correct
7 }
8
9 int ssl_socket_open2(SSL* ssl)
10 X509* cert = SSL_get-peer_certificate(ssl);
11 if (cert == NULL) {...}
12 long err = SSL_get_verify_result(ssl);
13 ...
14 return OK; // incorrect (missing check on err)
15 }
16
17 int ssl_socket_open3(SSL* ssl)
18 long err = SSL_get_verify_result(ssl);
19 if (err == X509_V_OK) {...}
20 X509* cert = SSL_get-peer_certificate(ssl);
21 if (cert) {...}
22 return OK; // correct
23 }
24
25 int ssl_socket_open4(SSL* ssl)
26 long err = SSL_get_verify_result(ssl);
27 switch (err) {
28 case X509_V_OK:
29 cert = SSL_get-peer_certificate(ssl);
30 }
31 return OK; // incorrect (missing check on cert)
32 }
```

Our goal is to learn a Datalog program that detects functions that misuse the OpenSSL API, whose behavior is defined as follows:

- 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 if no peer certificate was presented.

Functions should return OK only if (i) SSL_get-peer_certificate returns a non-null pointer, and (ii) SSL_get_verify_result returns the constant named X509_V_OK.

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

- OpSucc(l1, l2): Program control may flow from line l1 to l2.
- Check(x, l): The value of variable x is compared to a specific value at line l.
- Certify(x, l): Variable x at line l is assigned the return value of SSL_get-peer_certificate().
- Verify(x, l): Variable x at line l is assigned the return value of SSL_get_verify_result().
- Ok(l): The function that returns OK at line l correctly uses the OpenSSL API.

Relations OpSucc and Check are pre-defined as part of the program’s intermediate representation while relations Certify and
Syntax-Guided Synthesis of Datalog Programs

Verify can be automatically extracted from a given API, in this case OpenSSL. We provide an instance of these relations encoding the analyzed C program to ALPS, namely

\[
\text{Certify}(\text{cert}, 2), \text{Verify}(\text{err}, 3), \text{Check}(\text{cert}, 4), \text{Check}(\text{err}, 5), \ldots
\]

along with \(0k(6)\) and \(0k(22)\) as positive examples and \(0k(14)\) and \(0k(31)\) as negative examples in the output relation. ALPS generates the following program in 6 minutes.

\[
\begin{align*}
\text{CertFlow}(x, l) :& \text{Certify}(x, l), \text{OpSucc}(l, l_1, l_2). \\
\text{VeriFlow}(x, l) :& \text{Verify}(x, l), \text{OpSucc}(l, l_1, l_2). \\
\text{CertCheck}(l) :& \text{CertFlow}(x, l, \text{Check}(x, l), \text{OpSucc}(l, l_1, l_2)). \\
\text{VeriCheck}(l) :& \text{VeriFlow}(x, l, \text{Check}(x, l), \text{OpSucc}(l, l_1, l_2)). \\
0k(l) :& \text{CertCheck}(l), \text{VeriCheck}(l).
\end{align*}
\]

The rules are intended to be read right-to-left, with all variables universally quantified, and the \(\leftarrow\) operator interpreted as implication. Note that predicates \(\text{CertFlow}(x, l)\), \(\text{VeriFlow}(x, l)\), \(\text{CertCheck}(l)\), and \(\text{VeriCheck}(l)\) are not specified among the input or output relations; they are invented by ALPS, highlighting the rich space of programs it explores.\(^1\) We elaborate on how the search space is determined in Section 4.1. The relation \(\text{CertFlow}(x, l)\) (\(\text{VeriFlow}(x, l)\) resp.) indicates the return value of \(\text{SSL_get_verify_result}()\) flows to line \(l\). The relation \(\text{CertCheck}(l)\) (\(\text{VeriCheck}(l)\) resp.) means the return value of \(\text{SSL_get_verify_result}()\) compared to a specific value and control flows to line \(l\).

The Datalog program correctly captures an important portion of the proper use of the OpenSSL API. This example illustrates that ALPS represents a promising step towards synthesizing usable program analyzers. On the contrary, the state-of-the-art ILP tools Metagol and Zaatar fail to synthesize the program within 3 hours.

\section*{Example 2.2 (Relational queries).} Datalog is widely used as a relational query language due to its expressiveness and scalable performance [13, 27, 59]. ALPS can be used to synthesize sophisticated relational queries in Datalog from input-output behaviors.

We illustrate using ALPS to synthesize a relational query for finding students who take two different classes on the same day. The problem involves three input relations and one output relation with the following meaning:

- \(\text{Student}(s, n)\): Student \(s\) is associated with the ID \(n\).
- \(\text{Class}(c, d)\): Class \(c\) is held on day \(d\).
- \(\text{Enrolled}(n, c)\): The student having ID \(n\) is enrolled in class \(c\).
- \(\text{Busy}(s)\): Student \(s\) takes two different classes on the same day.

It is natural in a programming-by-example setting for the user to provide an instance specifying the input-output behavior of the desired query. Using such an instance comprising input relations regarding 14 students and 6 classes, and 5 examples in the output relation Busy, ALPS synthesizes the following Datalog program within 18 seconds:

\[
\begin{align*}
\text{EnrollClass}(n, c_1, l) :& \text{Enrolled}(n, c), \text{Class}(c, l). \\
\text{Busy}(s) :& \text{Student}(s, n), \text{EnrollClass}(n, c_1, l), \text{EnrollClass}(n, c_2, l), \text{c}_1 : \leftarrow \text{c}_2.
\end{align*}
\]

\(^1\)For readability, we provide intuitive names for invented predicates instead of mechanically generated ones by ALPS.

ESEC/FSE ’18, November 4–9, 2018, Lake Buena Vista, FL, USA

where \(\text{EnrollClass}\) is an invented predicate. While ostensibly simple, the above query is non-trivial to synthesize since it is semantically equivalent to the following complex SQL query:

\[
\begin{align*}
\text{SELECT} & \text{S.s FROM Student S} \\
\text{WHERE} & \text{S.n IN (SELECT E1.n FROM Enrolled E1, Enrolled E2, Class C1, Class C2 WHERE E1.n = E2.n AND E1.c \leftarrow E2.c AND E1.c = C1.c AND E2.c = C2.c AND C1.d = C2.d))}
\end{align*}
\]

In contrast, a state-of-the-art tool Scythe [69] for synthesizing SQL queries fails to generate the above SQL query within 3 hours.

Example 2.3 (Knowledge discovery). We demonstrate how ALPS synthesizes a prototypical program in knowledge discovery that is commonly used in the ILP literature: computing the transitive closure of a directed graph. The problem involves one input relation edge and one output relation 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, edge, with the following example graph:

\[
\text{\begin{itemize}
\item \text{edge}(1, 2) \\
\item \text{edge}(2, 3) \\
\item \text{edge}(3, 5) \\
\item \text{edge}(5, 7) \\
\item \text{edge}(4, 5) \\
\item \text{edge}(4, 7) \\
\item \text{edge}(4, 2) \\
\item \text{edge}(4, 1) \\
\end{itemize}}
\]

where an edge from node \(i\) to node \(j\) indicates that \(\text{edge}(i, j)\) appears in the input relation.

Given 4 examples of the output relation path, ALPS synthesizes the following recursive program in less than one second, which computes the transitive closure of a directed graph.

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

In fact, since ALPS maintains all possible programs, it also discovers the following \textit{non-linear} recursive program:

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

We next elaborate on three techniques that ALPS combines synergistically in order to realize this result.

\section*{Meta-rule-guided synthesis.} Learning Datalog programs is a complex task; even approximate learning is hard [18].\(^2\) To overcome this barrier, ALPS exploits the observation that in practice Datalog programs comprise rules with similar latent syntactic structure. Note that both of the recursive rules above are very similar: the only difference is in the relation names. We capture such similarities via the notion of \textit{meta-rules} [50] 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 meta-rule:

\[
\begin{align*}
\text{R}_0(x, z) :& \leftarrow \text{R}_1(x, y), \text{R}_2(y, z).
\end{align*}
\]

Many Datalog rules follow a similar chain pattern. This suggests a strategy for synthesizing Datalog programs: enumerate all possible instantiations of the above meta-rule with concrete relation

\(^2\)Datalog can in fact be viewed as augmenting relational algebra, which is widely used in the form of SQL, with recursion.

\(^3\)In approximate learning, the learnt program is not guaranteed to be consistent with the given examples.
names and examine their combinations. Furthermore, we can think of slight variations of this pattern to capture a broader range of programs. For example, the first rule in Example 2.2 is a slight variation of this meta-rule, the only difference being the arity of predicate \( \text{EnrollClass} \). We formalize this concept via a process called augmentation (see Section 4.4).

**Query-by-Committee (qbc).** ALPS uses an active learning technique to iteratively pose membership queries about the contents of the output relation, which can be answered upfront or in an interactive manner. To minimize the number of queries, at each step, ALPS picks a query that can prune the space of candidate programs to converge to the oracle’s desired program. In the above example, it begins by posing the query: *Is there a path from 1 to 2?* It obtains the answer yes. ALPS then poses the next query: *Is there a path from 3 to 2?* It obtains the answer no, and the process continues. After 4 queries out of 49 possible queries—all pairs \((i,j)\) where \(i,j \in [1,7]\), ALPS arrives at the above programs. In contrast, picking examples to query randomly may result in a large number of questions. For instance, in 100 trials with random selection, the maximum number of queries is 27, with an average of 12.

**Bidirectional search strategy.** To explore the search space efficiently, we propose a bidirectional synthesis strategy to maintain consistent candidate programs as a concise committee; and, using bi-directional search to efficiently represent and effectively refine all possible programs in the search space.

The above three techniques are combined in a synergistical manner in ALPS: meta-rules define a reasonably large and rich space of candidate programs; the syntactic structure of the space enables bidirectional search to efficiently represent and effectively refine all consistent candidate programs as a concise committee; and, using this committee, QBC guides the refinement by picking the most controversial example to query in the next iteration.

## 3 PROBLEM FORMULATION

In this section, we formalize Datalog and the synthesis problem.

### 3.1 Datalog Programs

**Rules.** A term \( t \) is either a variable \( x, y, z, \ldots \), or a constant \( a, b, c, \ldots \). A relation symbol \( p, q, r, \ldots \) is associated with an arity \( \text{ar}(r) \). An atom is an application of a relation symbol to a vector of variables and constants, e.g., \( r(x, y, a) \) for a relation \( r \) with arity 3. A ground atom is an application of a relation symbol to constants, e.g., \( r(a_1, \ldots, a_n) \), where \( a_i \) are constants. A Datalog rule \( C \) is an expression of the form:

\[
A \leftarrow B_1, B_2, \ldots, B_n.
\]

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 \( A \).

**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 \( B \) 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 input \( I \). 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 synthesize Datalog programs through examples. An example is a ground atom from the Herbrand base \( B \), 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 \( E^+ = \{e \in B | O(e) = +\} \) and \( E^- = \{e \in B | O(e) = -\} \) be the positive and negative examples defined by the oracle respectively. The goal is to find \( P \in \mathcal{H} \) such that: for all \( e \in E^0, P \cup I \models e \), and for all \( e \in E^-, P \cup I \not\models e \).

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 \( V_E \).

## 4 OUR APPROACH

In this section, we present the synthesis algorithm underlying ALPS. Section 4.1 describes the structure of the search space. Section 4.2 presents the algorithm parameterized by refinement operators. Section 4.3 instantiates the algorithm with meta-rules. Section 4.4 describes our methodology for designing meta-rules. Lastly, Section 4.5 states formal properties of our algorithm.

### 4.1 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. For our running example (Example 2.3), we consider a simple hypothesis space where all programs use a subset of the following four rules:

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

We denote the Datalog program consisting of rules \( r_1, r_2, r_3, r_4 \) as \( P_{ijk} \).

**Generality order.** We structure the search by imposing a generality order on the space of Datalog programs. To define this order, we use \( \theta \)-subsumption [54], which is a syntactic approach for deciding whether one rule subsumes (is more general than) another rule.

Formally, a rule \( C \) subsumes another rule \( D \) if there is a variable substitution \( \theta \) such that \( C \theta \) has the same head as \( D \), and all atoms in
that are construct committee forms which recalculate the most-general programs and the most-specific we are discarding only semantically equivalent programs. For our rule $C$ and $O$ always consistent with $E$ most-general programs pair $S$.

\[ \text{4.2 The ALPS Algorithm} \]

Given a synthesis problem $S = (H, O, I)$, ALPS applies Algorithm 1 to find a solution for $S$. It is a fixpoint algorithm that maintains a pair $E = (E^+, E^-)$ of positive and negative examples, and a set of most-general programs $\bar{P}$ and most-specific programs $P$ that are always consistent with $E$. The examples are initially empty, and $\bar{P}, P$ are initialized to be the most general and most specific programs respectively (we define this initialization in Section 4.3). 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 4.3). 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 $\bar{P}, P$ 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 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 [24, 62], a greedy yet effective strategy commonly used in active learning [61]. Since there are only two possible labels for an example, we use a simplified definition, which is essentially equivalent to disagreement count.

**Definition 4.1 (Vote entropy).** For an example $e$ and set of committee members $K$, the normalized vote entropy is:

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

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 and the query posed in each iteration for our running example.

\[ \text{4.3 Refinement with Meta-Rules} \]

We now give concrete definitions of the 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 [50], 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:

\[ R_1(x_1, \ldots, x_m) :: R_2(y_1, \ldots, y_{m_2}), \ldots, R_n(z_1, \ldots, z_{m_n}) \]

where $x_i, y_i, z_i \in V_1$ and $R_i \in V_2$. 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:

\[ T_1 : R_3(x, y) :: R_4(x, y) \]

\[ T_2 : R_4(x, z) :: R_5(y, z) \]

\[ T_3 : R_5(x, x) :: R_1(x, x) \]

\[ T_4 : R_6(x, z) :: R_7(y, y), R_8(y, z) \]

Similar to rules, a generality order between meta-rules can be established using $\theta$-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.
Meta-rule-guided refinement. Algorithm 2 describes our refine-
ment operations, $F^\dagger$ and $F^\dagger$, which are parameterized by a set of meta-rules $T$. We explain only top-down refinement $F^\dagger$ in detail, since bottom-up refinement $F^\dagger$ works in a symmetrical manner.

The algorithm begins with the given set of programs $F$. Then, it iteratively specializes the programs by applying the specialization operator $\rho^1$, which is guided by $T$ (line 2–5). In each iteration, the condition $F \not\subseteq V_E$ 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 $p(1, 2)$, all the more specific programs $P_{34}, P_2, P_3, P_4$ are eliminated from consideration as well.

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

The final piece of the puzzle is the specialization operator $\rho^1$. Here, $\rho^1$ can specialize a program in two ways: (1) replace a rule it 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 $r$.

4.4 Augmentation and Predicate Invention

The choice of meta-rules dictates the effectiveness of our synthesis algorithm. If the set of meta-rules is too large, then $\alphaLP$ 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 intuitive meta-rules that are specified manually (e.g., the chain meta-rule), 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, the fewer meta-rules will be generated from $T$.

As an example of augmentation, consider these two meta-rules:

$T_1 : R_0(y) :- R_1(z), R_2(y, z).

T_2 : R_0(y, z) :- R_3(z, x), R_2(y, z).

Figure 1: Version space in each iteration (red/yellow nodes represent most-general/specific programs in the current iteration; purple nodes represent programs that are both most general and most specific in the current iteration; and grey nodes represent programs that have been evaluated). An arrow from $u$ to $v$ means that program $u$ is more general than program $v$. 
Then, $T_2$ is an augmentation of $T_1$ with distance 2.

The augmentation distance required for ALPS to synthesize a program $P$ from an initial set of meta-rules $T$ is:

$$AD(P, T) = \max_{T_i \in T} \min_{T_j \in T} AD(T_i, T_j)$$

where $T_i$ ranges over all meta-rules used in $P_i$. In our experiments, we could synthesize almost all of the programs using an augmentation distance of 5 from three chain meta-rules.

**Predicate invention.** Another orthogonal way to relate the richness of programs in the search space is predicate invention. Predicate invention helps to break a complex rule into simpler ones, and thereby enables to reuse existing meta-rules. More importantly, it is unavoidable for Datalog programs with recursion. For instance, consider the following program which computes strongly connected components (SCC) in a directed graph:

```plaintext
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y), edge(y,z).
scc(x,y) :- path(x,y), path(y,x).
```

Here, the input and output relations are edge and scc, respectively. Given that scc cannot be derived by any set of clauses in terms of the input relation edge, a new predicate path must be invented. The difficulty with predicate invention lies in determining what form the invented predicates should take. Without meta-rules, we have no way to effectively constrain the syntax of such predicates. 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.

### 4.5 Properties of ALPS

The ALPS synthesis algorithm (Algorithm 1) always makes progress after every query to oracle $O$. We remove from consideration a controversial example from $B$. Since the set of possible examples $B$ is finite, the algorithm always terminates. It 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. We provide its proof in the Appendix.

**Theorem 4.2.** Let $S = (H, O, I)$ be a synthesis problem such that a solution to $S$ exists in $H$. Let $P$ be the output of ALPS. Then:

1. (Soundness) Every $P \in P$ is a solution to $S$.
2. (Completeness) For every solution $P \in H$ to $S$, there exist programs $P_1, P_2 \in P$ 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 a variety of synthesis tasks from different domains. Our implementation\footnote{Our artifact is available on GitHub: https://github.com/XujieSi/fse18-artifact-183.} comprises about 8,000 lines of C++ code. It uses the fixpoint engine of the Z3 smt solver [31] for Datalog evaluation. Our experiments were performed on a Linux machine with 16 GB of RAM and a 3.0 GHz processor.

---

### Table 1: Benchmark characteristics.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Brief description</th>
<th>#Relations</th>
<th>#Rules</th>
<th>Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Knowledge Discovery</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inflammation</td>
<td>diagnosis of bladder inflammation</td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>abduce</td>
<td>grandparent of given father/mother [46]</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>animals</td>
<td>distinguishing classes of animals [46]</td>
<td>13</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ancestor</td>
<td>ancestor in a family tree [50]</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>buildall</td>
<td>learn a stable wall strategy [50]</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>samegen</td>
<td>same generation in a family tree [7]</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>path</td>
<td>all-pairs reachability in directed graph</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>scc</td>
<td>compute SCCs in directed graph</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>Program Analysis</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>polysite</td>
<td>polymorphic call-site inference for Java</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>downcast</td>
<td>downcast safety checker for Java</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>rv-check</td>
<td>return-value-checker in ALPSan [72]</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
| andersen | inclusion-based pointer analysis for C [12] | 5 | 4 | ✔
| 1-call-site | 1-call-site pointer analysis for Java [70] | 9 | 4 | |
| 2-call-site | 2-call-site pointer analysis for Java [70] | 9 | 4 | |
| 1-object | 1-object-sensitive pointer analysis [42] | 11 | 4 | |
| 1-type | 1-type-sensitive pointer analysis [66] | 12 | 4 | |
| 1-obj-type | 1-type-1-object sensitive analysis [66] | 13 | 5 | |
| escape | escape analysis for Java | 10 | 6 | |
| modref | mod-ref analysis for Java | 13 | 10 | |
| **Relational Queries** | | | | |
| sql-1 - 15 | SQL queries [69] | %7 | %4 | |

Our empirical evaluation aims to answer the following questions:

**Q1.** How does ALPS perform on synthesis tasks from a variety of domains in terms of synthesis time and number of queries?

**Q2.** How much does meta-rule augmentation speed up synthesis?

**Q3.** How effective is QBC in reducing the number of queries asked?

**Q4.** How sensitive is ALPS to changes in a given input data?

**Q5.** How does ALPS compare with existing synthesis techniques?

### 5.1 Benchmark Suite

We collected 34 synthesis tasks from three different application domains: (i) knowledge discovery, (ii) program analysis and (iii) relational queries. Table 1 presents useful characteristics of these benchmarks. The last three columns show the number of input–output relations, the number of rules of the smallest desired program, and whether the desired program is recursive or not, respectively.

**Knowledge discovery.** The knowledge discovery benchmarks comprise 8 tasks of synthesizing Datalog programs frequently used in the artificial intelligence and database literature. The goal of the first benchmark inflammation is to discover interesting correlations between patient risk factors and a disease called acute inflammations of urinary bladder. We used a dataset created by a medical expert to enable expert systems that perform presumptive diagnosis of the disease [20].\footnote{Available at http://archive.ics.uci.edu/ml/datasets/Acute+Inflammations.} The next four benchmarks (abduce, ancestor, animals, and buildall) are widely used in the field of inductive logic programming [46, 50]. The samegen benchmark is a standard Datalog program in the database literature [7]. The path benchmark is the problem described in Example 2.3 and the SCC benchmark is the problem of computing strongly connected components in a directed graph.
Program analysis. The program analysis benchmarks comprise 11 tasks of synthesizing static analyzers written in Datalog:

- **polsite** is a polymorphic call-site inference analyzer for Java;
- **downcast** is a downcast safety checker for Java;
- **rv-check** is the static API misuse detector described in Example 2.1, which is motivated from a return value checker used in a tool called APIsan [72]. APIsan identifies API misuses by detecting inconsistent uses of the return values of API functions. However, the tool is neither sound nor complete due to the limitation of its statistical method. This observation motivated our rule-based approach for static API misuse detection.
- **andersen** is a classic pointer analysis for C [12];
- The next five benchmarks are pointer analyzers for Java with various context abstractions [42, 66, 70].
- **modref** is a mod-ref analysis for Java and **escape** is an escape analysis for Java. Both benchmarks originated from a programming assignment in an online course on program analysis [5].

Relational queries. These benchmarks comprise 15 synthesis tasks from Stack Overflow posts and textbook examples [69]. We chose the 15 tasks of synthesizing SQL queries that can be expressed in Datalog. Each task involves up to 6 input tables and one output table. The desired Datalog programs comprise up to four rules.

5.2 Experimental Setup

Meta-rules. We first apply only the following three chain meta-rules with up to 5 augmentations for all benchmarks:

\[
R_0(v_1, v_2) \leftarrow R_1(v_1, v_2).
\]

\[
R_0(v_1, v_3) \leftarrow R_1(v_1, v_2), R_2(v_2, v_3).
\]

\[
R_0(v_1, v_4) \leftarrow R_1(v_1, v_2), R_2(v_2, v_3), R_3(v_3, v_4).
\]

We observe that **alps** fails to synthesize five context-sensitive pointer analysis benchmarks, as the necessary augmentation distance is too far and offers no filtering of the search space. In these cases, we also include domain specific meta-rules, e.g. meta-rules extracted the remaining four benchmarks. Also, we set the maximum number of invented predicates to 4.

Input-output relations. For each benchmark, **alps** begins with no examples of output relations and iteratively poses membership queries about the contents of the output relations. To answer such queries, we used the known solution of the benchmark (i.e., the desired Datalog program) as an oracle. The populated input relations range in size from 3 to 77 with an average of 22.

5.3 Evaluation Results

Q1: Number of queries and synthesis time. Table 2 presents 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 takes 25 seconds to synthesize 3 programs; and, in the process, it evaluates 24,280 programs out of 10^16 programs in the search space.

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

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. Only two benchmarks can be synthesized with no augmentation, while most benchmarks (29 out of 34) 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.

Starting with only chain meta-rules, however, **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 of the five context-sensitive pointer analysis benchmarks, as shown in Table 2.1

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 maximizes vote entropy. For each benchmark, we ran 100 trials with random example selection. Figure 2(a) shows a box plot of the number of queries made in these trials. We see that random selection, on average, performs much worse than **qbc**. For instance, on the ancestor benchmark, the median for random selection is more than 120 queries, while **qbc** only needs 11.

Compared to the best-case scenarios of random selection, **qbc** makes roughly the same amount (about /-5 queries) of queries for most benchmarks. For the **andersen** benchmark, one random trial behaves surprisingly well: it only requires 8 questions to synthesize all 4 expected rules, one of which eliminates 99.7% of the committee members. This is an artifact of **qbc**’s conservative example-selection approach: it prefers high-entropy examples, so it may miss a low-entropy example that could eliminate most of the committee, indicating **qbc** is not optimal but practically effective.

Q4: Sensitivity to the size of input data. We now investigate the effects of increasing the size of input data on the number of queries needed. We focus on the **andersen** benchmark, as it is demonstrative of the behavior across our benchmark suite. Recall that **andersen** is a pointer analysis for C programs. We systematically increased the size of the input data (i.e., the size of the analyzed program), and measured the number of queries needed to synthesize Andersen’s analysis, as well as the synthesis time. Figure 2(b) summarizes the results of this experiment. It shows that as the size of the data increases, the number of questions asked stays roughly constant. As a result, the question ratio (number of questions asked to the total number of possible questions) is reduced by approximately 0.01. For instance, on the average, **alps** asks a very small number of queries for them.

- **1** obj - type takes **alps** 17 hours to finish and hence is marked as timeout.
- **2** We skip the SQL benchmarks as **alps** asks a very small number of queries for them.
<table>
<thead>
<tr>
<th></th>
<th>#queries asked</th>
<th>#possible queries</th>
<th>#synthesized programs</th>
<th>#evaluated programs</th>
<th>search space</th>
<th>total time (sec.)</th>
<th>Metagol run. time (sec.)</th>
<th>Zaatar run. time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>inflammation</td>
<td>8</td>
<td>120</td>
<td>4</td>
<td>2327</td>
<td>$10^6$</td>
<td>4.3</td>
<td>0.51</td>
<td>0.47</td>
</tr>
<tr>
<td>abduce</td>
<td>11</td>
<td>392</td>
<td>1</td>
<td>4613</td>
<td>$10^6$</td>
<td>3.36</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>animals</td>
<td>16</td>
<td>80</td>
<td>2</td>
<td>45152</td>
<td>$10^6$</td>
<td>75.8</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>ancestors</td>
<td>11</td>
<td>450</td>
<td>3</td>
<td>24820</td>
<td>$10^{38}$</td>
<td>24.6</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>buildWall</td>
<td>12</td>
<td>392</td>
<td>13</td>
<td>61654</td>
<td>$10^6$</td>
<td>128.7</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>samegen</td>
<td>12</td>
<td>162</td>
<td>2</td>
<td>110338</td>
<td>$10^6$</td>
<td>22.3</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>path</td>
<td>4</td>
<td>49</td>
<td>3</td>
<td>384</td>
<td>$10^6$</td>
<td>0.26</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sec</td>
<td>14</td>
<td>128</td>
<td>4</td>
<td>57013</td>
<td>$10^6$</td>
<td>88.7</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>polysite</td>
<td>14</td>
<td>204</td>
<td>5</td>
<td>27432</td>
<td>$10^22$</td>
<td>130.0</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>downcast</td>
<td>9</td>
<td>912</td>
<td>1</td>
<td>56489</td>
<td>$10^{28}$</td>
<td>299.8</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>rv-check</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>393740</td>
<td>$10^{38}$</td>
<td>361.5</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>andersen</td>
<td>15</td>
<td>64</td>
<td>1</td>
<td>100345</td>
<td>$10^{38}$</td>
<td>148.0</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>1-call-site</td>
<td>14</td>
<td>152</td>
<td>3</td>
<td>99697</td>
<td>$10^22$</td>
<td>178.3</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>2-call-site</td>
<td>17</td>
<td>672</td>
<td>1</td>
<td>184024</td>
<td>$10^22$</td>
<td>601.8</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>1-object</td>
<td>18</td>
<td>655</td>
<td>1</td>
<td>93362</td>
<td>$10^{48}$</td>
<td>705.1</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>1-type</td>
<td>13</td>
<td>215</td>
<td>2</td>
<td>10038</td>
<td>$10^{38}$</td>
<td>21.6</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>1-obj-type</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>escape</td>
<td>9</td>
<td>40</td>
<td>12</td>
<td>5706</td>
<td>$10^{12}$</td>
<td>9.9</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>modref</td>
<td>14</td>
<td>145</td>
<td>1</td>
<td>1346754</td>
<td>$10^{38}$</td>
<td>5307</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-1</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>30</td>
<td>$10^6$</td>
<td>0.07</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-2</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>7</td>
<td>$10^6$</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>$10^6$</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-4</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>19</td>
<td>$10^2$</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-5</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>$10^6$</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-6</td>
<td>6</td>
<td>729</td>
<td>1</td>
<td>44</td>
<td>$10^2$</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-7</td>
<td>1</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>$10^6$</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-8</td>
<td>5</td>
<td>48</td>
<td>2</td>
<td>230</td>
<td>$10^{38}$</td>
<td>1.60</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-9</td>
<td>5</td>
<td>27</td>
<td>1</td>
<td>9</td>
<td>$10^{38}$</td>
<td>0.30</td>
<td>timeout</td>
<td>0.01</td>
</tr>
<tr>
<td>sql-10</td>
<td>8</td>
<td>40</td>
<td>1</td>
<td>778</td>
<td>$10^{12}$</td>
<td>63.2</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-11</td>
<td>4</td>
<td>25</td>
<td>6</td>
<td>1192</td>
<td>$10^{12}$</td>
<td>1.86</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-12</td>
<td>4</td>
<td>34</td>
<td>1</td>
<td>117</td>
<td>$10^{12}$</td>
<td>0.20</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-13</td>
<td>3</td>
<td>80</td>
<td>1</td>
<td>4</td>
<td>$10^{38}$</td>
<td>0.01</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-14</td>
<td>3</td>
<td>651</td>
<td>1</td>
<td>13</td>
<td>$10^{22}$</td>
<td>90.9</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>sql-15</td>
<td>5</td>
<td>266</td>
<td>1</td>
<td>344</td>
<td>$10^{22}$</td>
<td>17.7</td>
<td>timeout</td>
<td>timeout</td>
</tr>
</tbody>
</table>

Table 2: ALPS performance results (the timeout limit is 3 hours).

Figure 2: (a) Box plot of the number of queries asked by random selection (green dots mark the number of queries by qbc); and (b) Number of queries asked by ALPS for the andersen benchmark under different sizes of input data (where X=7).

significantly, from nearly 25% to 0.002%. The number of evaluated programs also remains roughly constant. As expected, the synthesis time increases with more data, as ALPS needs to invoke the Datalog solver on larger inputs.

Q5: Comparison with other tools. We now compare ALPS with two state-of-the-art ILP tools: Metagol [19] and Zaatar [9]. We supply both of these tools with all ground facts upfront since they are non-interactive.

Metagol is an ILP tool that is an instance of the meta-interpretive learning framework [50], 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 fails 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 [9] 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.

Summary. To summarize, our experimental evaluation demonstrates (i) the ability of ALPS to synthesize sophisticated algorithms; (ii) the effectiveness of meta-rules and augmentations; (iii) the importance of 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 approach. We outline these next along with proposals to mitigate them.

- ALPS may fail to synthesize a desirable program because the input relations do not cover all corner cases (i.e., overfitting). We can mitigate this threat by allowing the user to provide a large input and 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.
- The labeled examples may be noisy. This threat can be mitigated by collecting answers from multiple oracles (e.g., through crowdsourcing or multiple reference implementations) and using the majority vote as the final answer.
- A large number of equivalent final programs exist due to many applicable combinations of meta-rules. This threat can be mitigated by sacrificing completeness and reporting a subset of programs.
- The meta-rules generated by ALPS may be insufficient to capture a desired program. This threat can be mitigated by using a larger number of invented predicates. The more such predicates we use, the simpler rules we obtain. Eventually, the desired program will comprise typical rules, thus enabling ALPS to synthesize it.

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 [21], from vast amounts of mined data, e.g., biological data [47]. 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, FIP [15] poses existential queries and CGOL [48] 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 [29]. Meta-Interpretive Learning has been used to learn Prolog programs for string manipulation tasks [38]. 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 [19, 64, 67, 68]. At a high-level, meta-rules can also be seen as program sketches [67], where the holes are the relation symbols.

Interactive synthesis by example. Some of the programming-by-example (PBE) approaches interactively query an oracle for examples. Jha et al. [32] present an oracle-guided synthesis procedure for straight-lines 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 [37] to learn a grammar.

Synthesis of recursive programs. A number of works have targeted the problem of synthesizing recursive programs [8, 23, 34, 35, 53, 55]. 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. [36], and more recently in Flash-Fill [28] for spreadsheet manipulation. ALPS maintains a version space using most-general and most-specific programs, as first proposed by Mitchell [43]. Our setting is different than Mitchell’s in the sense that the search space is determined by a set of meta-rules that forms a partially ordered set, and thereby we can exploit the generality order on the meta-rules for our bidirectional search.

Learning for program analysis. Recently, several systems have been created that apply machine learning techniques to program analysis. Oh et al. [52] use Bayesian optimization techniques to effectively learn adaptation strategies for parametric program analysis, while Bielik et al. [16] 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 of the program analysis, expressed in Datalog.

7 CONCLUSIONS AND FUTURE WORK

We proposed a programming-by-example system, ALPS, to synthesize Datalog programs. It employs a syntax-guided synthesis approach that prunes the search space by exploiting the observation that Datalog programs in practice comprise rules with similar latent syntactic structure. To this end, we synergistically combined three techniques: a novel approach to systematically generate meta-rules, taking advantage of domain knowledge; an active learning technique called query-by-committee to minimize the number of examples needed; and a bidirectional synthesis strategy to explore the search space efficiently. We evaluated the system on a variety of synthesis tasks from different domains and demonstrated that it significantly outperforms existing state-of-the-art tools.

We envision many useful directions to extend our work. One direction concerns repairing existing Datalog programs using examples rather than synthesizing them from scratch. For instance, a user may modify an existing program analysis by refining its rules to be consistent with known true and false positives. Another direction concerns using probabilistic models to expedite the search, for instance, by predicting which augmentations to use. Finally, we plan to explore how to synthesize programs that are not only correct with respect to examples, but also optimal with respect to objectives such as likelihood or evaluation complexity.

A PROOF OF ALPS PROPERTIES

To prove Theorem 4.2, we use two key properties about the interplay between QBC and bidirectional search. The following lemma captures the fact that the algorithm does not miss any controversial examples, and thus always makes progress in terms of pruning the search space.

**Lemma A.1.** Let \( P \subseteq H \) and \( e \in B \). Then \( D(e, P) \neq 0 \) if and only if there exist programs \( P_1, P_2 \in P \) such that
\[
P_1 \cup I \models e \quad \text{and} \quad P_2 \cup I \not\models e
\]

**Proof.** This is directly implied by the definition of vote entropy (see Definition 4.1).

The next lemma states key invariants that hold at every iteration of the algorithm: (i) It does not miss any programs that are solutions to the synthesis problems, by ensuring that the contours of the version space form an upper/lower bound of every solution. (ii) It ensures that if the current version space contains non-solutions, then there are non-zero entropy examples we can ask the oracle that can eliminate them.

**Lemma A.2 (Invariant).** Let \( S = (H, O, I) \) be a synthesis problem such that a solution to \( S \) exists in \( H \). Let \( E = (E^+, E^-) \) be the set of known examples at any point during execution, and let \( P = \overline{P} \cup P \).

Then:

1. For every solution \( P' \in H \) to \( S \), there exist programs \( P_1, P_u \in P \) such that \( P_1 \subseteq P' \subseteq P_u \).

2. If there exists a program \( P \in P \) that is not a solution to \( S \), then \( \exists e \in B. D(e, P) \neq 0 \).

**Proof.** We first show item (1). First, notice that every solution \( P' \in H \) to \( S \) belongs in the version space \( V_\epsilon \), since it satisfies all current examples. Since \( P \supseteq \max(V_\epsilon) \) (by the definition of \( F^+ \) in Algo. 1), there exists \( P_u \in P \) such that \( P' \subseteq P_u \). Similarly, since \( P \supseteq \min(V_\epsilon) \), there exists \( P_1 \in P \) such that \( P_1 \subseteq P' \).

We next show item (2). Suppose that \( P \in P \) is not a solution to \( S \), and let \( P' \) be a solution to \( S \). Then, there exists an example \( e \in B \) such that either (a) \( P \cup I \models e \) and \( P' \cup I \not\models e \), or (b) \( P \cup I \not\models e \) and \( P' \cup I \models e \). We now distinguish two different cases:

- Case (a) holds: since \( P' \) is a solution, item (1) tells us that there exists \( P_0 \in P \) such that \( P' \subseteq P_0 \). This implies that \( P_u \cup I \not\models e \). Because now \( P, P_u \) disagree on example \( e \), Lemma A.1 implies that \( D(e, P) \neq 0 \).

- Case (b) holds: since \( P' \) is a solution, item (1) tells us that there exists \( P_1 \in P \) such that \( P_1 \subseteq P' \). This implies that \( P_1 \cup I \models e \). Because now \( P, P_1 \) disagree on example \( e \), Lemma A.1 implies that \( D(e, P) \neq 0 \).

Thus, in both cases we find an example \( e \) such that \( D(e, P) \neq 0 \).

Equipped with the Lemmas A.1 and A.2, we are now ready to prove Theorem 4.2.

**Proof.** The algorithm terminates when for every example \( e \in B \), we have \( D(e, P) = 0 \). Recall that \( P = \overline{P} \cup P \), where \( E = (E^+, E^-) \) are the known examples.

- (Soundness) As \( \forall e \in B. D(e, P) = 0 \), the contrapositive of item (2) of Lemma A.2 indicates that every \( P \in P \) is a solution.
- (Completeness) This holds directly from item (1) of Lemma A.2.
- (Termination) At every iteration, the algorithm adds one example to either \( E^+ \) or \( E^- \). Notice that, after an example \( e \) is added to \( E = (E^+, E^-) \), it cannot be added again, since it will never be controversial \( D(e, P) \neq 0 \) from that point on. Since we have finitely many examples in \( B \), the algorithm terminates after finitely many steps.

This concludes the proof of the theorem.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback. This research was supported by DARPA under agreement #FA8750-15-2-0009 and by NSF awards #1253867, #1526270, and #1652140.


[73] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective Interactive Resolution of Static Analysis Alarms. In OOPSLA.