Provenance-Guided Synthesis of Datalog Programs

MUKUND RAGHOTHAMAN, University of Southern California, USA
JONATHAN MENDELSON, University of Pennsylvania, USA
DAVID ZHAO, University of Sydney, Australia
MAYUR NAIK, University of Pennsylvania, USA
BERNHARD SCHOLZ, University of Sydney, Australia

We propose a new approach to synthesize Datalog programs from input-output specifications. Our approach leverages query provenance to scale the counterexample-guided inductive synthesis (CEGIS) procedure for program synthesis. In each iteration of the procedure, a SAT solver proposes a candidate Datalog program, and a Datalog solver evaluates the proposed program to determine whether it meets the desired specification. Failure to satisfy the specification results in additional constraints to the SAT solver. We propose efficient algorithms to learn these constraints based on "*why*" and "*why not*" provenance information obtained from the Datalog solver. We have implemented our approach in a tool called ProSynth and present experimental results that demonstrate significant improvements over the state-of-the-art, including in synthesizing invented predicates, reducing running times, and in decreasing variance in synthesis performance. On a suite of 40 synthesis tasks from three different domains, ProSynth is able to synthesize the desired program in 10 seconds on average per task—an order of magnitude faster than baseline approaches—and takes under a second for 28 of them.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; **Constraint and logic languages**; • **Information systems** → **Relational database query languages**; • **Theory of computation** → **Data provenance**; • **Hardware** → **Theorem proving and SAT solving**.

Additional Key Words and Phrases: Program synthesis, Syntax-Guided Synthesis (SyGuS), Datalog, Counter-Example Guided Inductive Synthesis (CEGIS), data provenance, SAT solvers

ACM Reference Format:
Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-Guided Synthesis of Datalog Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 62 (January 2020), 27 pages. https://doi.org/10.1145/3371130

1 INTRODUCTION

The problem of synthesizing logical rules from data has important theoretical and practical implications in machine learning and program synthesis. Datalog [Abiteboul et al. 1994], a declarative logic programming language, has emerged as a popular medium for studying this problem due to its rich expressivity and scalable performance.

A variety of different techniques have been proposed for synthesizing Datalog programs, including meta-interpretive learning [Muggleton et al. 2015], reverse entailment [Muggleton 1995],

Authors' addresses: Mukund Raghothaman, University of Southern California, USA, raghotha@usc.edu; Jonathan Mendelson, University of Pennsylvania, USA, jonom@seas.upenn.edu; David Zhao, University of Sydney, Australia, dzha3983@uni.sydney.edu.au; Mayur Naik, University of Pennsylvania, USA, mhnaik@cis.upenn.edu; Bernhard Scholz, University of Sydney, Australia, bernhard.scholz@sydney.edu.au.

2475-1421/2020/1-ART62
https://doi.org/10.1145/3371130

version-space search [Si et al. 2018], SMT constraint solving [Albarghouthi et al. 2017], and numerical relaxation [Si et al. 2019]. Despite significant strides, however, all of these approaches are hindered by poor scalability, high variance in running times, or limited ability to handle expressive features such as recursion and invented predicates.

At the same time, query provenance [Cheney et al. 2009; Green et al. 2007a] has emerged as a powerful mechanism to enable a variety of tools that require meta-reasoning over Datalog programs, including debugging query results [Karvounarakis et al. 2010], counterexample-guided abstraction refinement (CEGAR) in static analyses [Zhang et al. 2014], and confidence computation in uncertain and probabilistic databases [Sarma et al. 2008].

The central insight of this paper is that provenance can also play a key role in program synthesis. We demonstrate this by proposing a provenance-guided approach to synthesize Datalog programs. In most guess-and-check approaches to program synthesis, such as counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006], the main challenge lies in identifying the reason for the failure of a particular candidate solution. Formal models of query provenance form the ideal template to structure such reasoning about failures.

Our approach follows the CEGIS paradigm: in each iteration, a SAT solver generates a candidate Datalog program, and a Datalog solver evaluates the generated program to determine whether it meets the desired input-output specification. In this context, our approach can also be regarded as an instantiation of the classic DPLL(T) procedure for automated theorem proving [Nieuwenhuis et al. 2005], with T being the theory of least fixed points.

A candidate Datalog program can fail to meet the desired specification in one of two ways: either by producing an *undesirable* output tuple or by failing to produce a *desirable* output tuple. Our approach handles both cases via additional constraints to the SAT solver in the next CEGIS iteration. Constraints encoding an erroneous derivation of an undesirable output tuple can be obtained directly via classical models of "*why*" provenance. However, reflecting on the non-derivation of a desirable output tuple leads to difficult ontological questions. We propose two new techniques to address this problem of "*why-not*" provenance: the first is a version of the delta-debugging algorithm that significantly strengthens the constraints from a non-derivation failure, and the second is a notion of co-provenance which identifies necessary constraints before the occurrence of a non-derivation failure.

In summary, our approach leverages provenance information from the Datalog solver in order to constrain the SAT solver in each CEGIS iteration. Conceptually, it constitutes a new approach to boolean function learning, where the target concept is the formula which encodes exactly the set of solutions to the synthesis problem. In practice, this provenance-guided approach is central to scaling synthesis and reducing variability in synthesis time—problems that plague existing approaches due to a large number of non-deterministic choices in the search process.

We have implemented our approach in a tool called ProSynth and demonstrate that it significantly improves over existing approaches, including in synthesizing invented predicates, reducing running times, and in decreasing variances in synthesis performance. In particular, we compare ProSynth to two state-of-the-art approaches: ALPS [Si et al. 2018], which uses a version-space search approach, and Difflog [Si et al. 2019], which uses an approach based on numerical relaxation. On a suite of 40 synthesis tasks from three different domains—knowledge discovery, program analysis, and relational queries—ProSynth is able to synthesize the desired program in 10 seconds on average per task, taking only under a second each for 28 of them. In contrast, ALPS times out in one hour on six tasks and takes 142 seconds on average for the rest. Likewise, Difflog times out on three tasks and takes 136 seconds on average for the rest. Finally, compared to Difflog, ProSynth exhibits much lower variability in running times across 32 runs on each task; and for all but three tasks, the maximum running time of ProSynth is lower than the median running time of Difflog.

We summarize the main contributions of this paper:

- We present a novel approach to synthesize Datalog programs from input-output specifications. It follows the CEGIS paradigm and leverages efficient off-the-shelf solvers—a SAT solver that guesses the candidate Datalog program and a Datalog solver that checks whether it meets the desired specification.
- We develop a general framework to harness provenance information from the Datalog solver to learn the constraints to the SAT solver in each CEGIS iteration. Our framework constitutes a new approach to learn boolean functions, and it is central to scaling synthesis and reducing variability in synthesis time.
- We demonstrate the effectiveness of our approach in a tool called ProSynth on a variety of synthesis tasks. ProSynth is able to synthesize more programs than state-of-the-art approaches and runs an order of magnitude faster, often in under a second.

The rest of the paper is organized as follows. Section 2 provides an illustrative overview of our approach. Section 3 formalizes the Datalog synthesis problem. Section 4 describes our synthesis framework and proves its correctness. Section 5 presents our empirical evaluation. Section 6 surveys related work and Section 7 concludes with a note on future directions.

2 MOTIVATING EXAMPLE

In this section, we illustrate the approach underlying ProSynth using an example synthesis task: computing strongly connected components in a given directed graph. We start with the specification of the problem, describe the workflow of ProSynth, and highlight the crucial role played by provenance-guided synthesis.

2.1 Problem Specification

We present the specification for the example task in Figure 1. Notice that this specification follows the *syntax-guided* formulation of synthesis problems (SyGuS) [Alur et al. 2013], and consists of two components: (*a*) a *semantic specification* in the form of relational input-output data, which in our example consists of an input relation edge describing the adjacency relation of the given directed graph and an output relation scc specifying the strongly connected components in it; and (*b*) a *syntactic specification* in the form of candidate rules, each of which is a Horn clause. We describe a procedure for generating candidate rules from a given relational schema in Section 3.3. While this procedure yields 166 candidate rules for the benchmark problem, for the sake of exposition, we will only consider a subset of 8 of them, denoted $r_0, r_1, \ldots, r_7$ in Figure 1. Each rule is a universally quantified first-order logical formula. For example, rule r_2 states that for every triple of nodes x, y, z, whenever the relation inv contains the tuple (x, y) and the relation edge contains the tuple (y, z), then the relation scc contains the tuple (x, z). An intermediate relation such as inv which is not specified in either the input or the output of the synthesis specification is called an *invented predicate*.

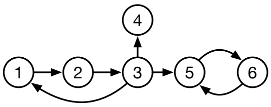
candidate rule generation

Given the set of input tuples I, the expected set of output tuples T_{exp}, and the set P_{all} of candidate rules, the goal is to efficiently find a subset of those rules, $P \subseteq P_{all}$, such that evaluating the Datalog program denoted by P on input I produces the output T_{exp}. This problem is NP-hard (see Theorem 3.2), and practical problem instances require a large number of candidate rules as well as the ability to discover one or more unlabeled predicates such as inv. In our example, ProSynth synthesizes the following correct program:

$$r_3 : \quad \texttt{scc}(x, y) \text{ :- } \texttt{inv}(x, y), \texttt{inv}(y, x).$$

Input tuples (I):

edge(1,2), edge(2,3),
edge(3,1), edge(3,4),
edge(3,5), edge(5,6),
edge(6,5)



(a) I.

Output tuples (T_{exp}):

scc(1,1), scc(5,5),
scc(2,2), scc(6,6),
scc(3,3), scc(5,6),
scc(1,2), scc(6,5),
scc(2,3),
scc(3,1),
scc(2,1),
scc(3,2),
scc(1,3)

(b) T_{exp}.

Candidate rules (P_{all}):

r_0 : scc(x, y) :- edge(x, y).
r_1 : inv(x, y) :- edge(x, y).
r_2 : inv(x, z) :- inv(x, y), edge(y, z).
r_3 : scc(x, y) :- inv(x, y), inv(y, x).
r_4 : scc(x, y) :- inv(x, y).
r_5 : scc(x, y) :- inv(y, x).
r_6 : inv(x, y) :- edge(y, x).
r_7 : inv(x, z) :- inv(x, y), edge(z, y).

(c) P_{all}.

Fig. 1. An example specification for Datalog program synthesis. The desired target program accepts as input a directed graph (represented by the edge relation) and outputs information about its strongly connected components (represented by the scc relation). The set of rules in the target program can be any subset of the candidate rules labeled r_0–r_7.

r_6 : inv(x, y) :- edge(y, x).
r_7 : inv(x, z) :- inv(x, y), edge(z, y).

Somewhat unexpectedly, this solution, $P_1 = \{r_3, r_6, r_7\}$, encodes the intermediate concept of a reverse path, that is, inv contains tuple (x, y) if and only if there is a path from y to x. Alternatively, ProSynth could have synthesized the following equivalent program, $P_2 = \{r_3, r_1, r_2\}$, which relies on the more conventional intermediate concept of a (forward) path:

r_3 : scc(x, y) :- inv(x, y), inv(y, x).
r_1 : inv(x, y) :- edge(x, y).
r_2 : inv(x, z) :- inv(x, y), edge(y, z).

2.2 Workflow of Our Approach

We now describe the solution workflow of ProSynth. Recall that the underlying synthesis algorithm follows the CEGIS paradigm. In Figure 2, we graphically present the interaction between the SAT solver and the Datalog solver in each CEGIS iteration. The algorithm maintains a boolean formula φ, which contains boolean variables v_r for candidate rules r in P_{all}. Each satisfying assignment computed by the SAT solver can be viewed as partitioning the rules into two sets, P^+ and P^-, consisting of rules whose corresponding boolean variables are respectively set to true and false by the SAT solver. We refer to φ as the *synthesis constraint*, and regard P^+ as the *candidate program*.

Initially, φ is set to TRUE, and the SAT solver can therefore return any subset of the candidate rules as the candidate program P^+. The Datalog solver evaluates this program on the given input I, and ProSynth determines whether the produced output $T_{out} = P^+(I)$ matches the desired output T_{exp}. If yes, the process terminates with P^+ as the desired program; otherwise, it uses the provenance information computed by the Datalog solver to strengthen φ (as described below), and the process is repeated. Note that the tuples $t \in T_{out}$ are controlled indirectly via the rule set P^+. A tuple t exists in the T_{out} if there exists a derivation tree for t whose rules are contained in P^+. Conversely, a tuple t does not exist in T_{out} if for all derivations tree some rules are not contained in P^+.

When $T_{out} \neq T_{exp}$, ProSynth computes provenance information for each mislabeled output tuple t, based on the following two cases, to obtain a boolean formula ($\neg\psi$ in the first case and ω in the second case) that strengthens ϕ:

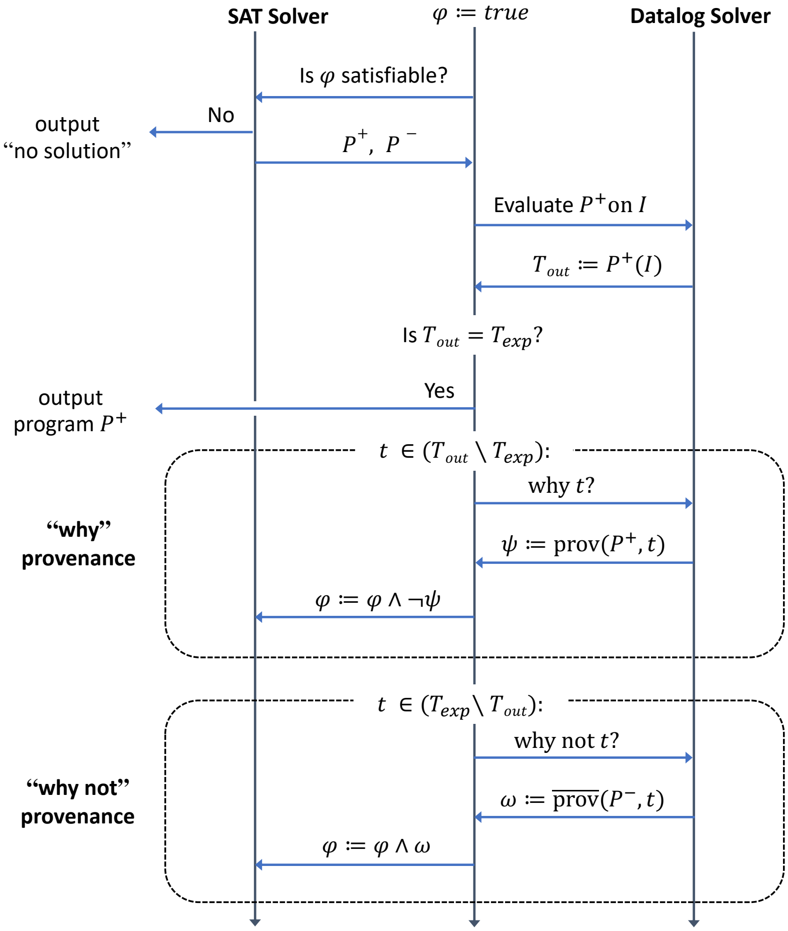

SAT Solver
$\varphi \coloneqq true$
Datalog Solver
Is φ satisfiable?
No
output "no solution"
P^+, P^-
Evaluate P^+ on I
$T_{out} \coloneqq P^+(I)$
Is $T_{out} = T_{exp}$?
Yes
output program P^+
$t \in (T_{out} \setminus T_{exp})$:
why t?
"why" provenance
$\psi \coloneqq \mathrm{prov}(P^+, t)$
$\varphi \coloneqq \varphi \land \neg\psi$
$t \in (T_{exp} \setminus T_{out})$:
why not t?
"why not" provenance
$\omega \coloneqq \overline{\mathrm{prov}}(P^-, t)$
$\varphi \coloneqq \varphi \land \omega$

Fig. 2. Message sequence chart depicting the interaction between the SAT solver and the Datalog solver in each CEGIS iteration of ProSynth.

- If $t \in T_{out} \setminus T_{exp}$, i.e., t is an undesirable tuple that was derived, ProSynth leverages existing notions of *"why"* provenance to compute ψ. Intuitively, ψ aims to disable certain rules in P^+ (i.e., move them to P^-) to prevent the derivation of t. false positive
- If $t \in T_{exp} \setminus T_{out}$, that is, t is a desirable tuple that was not derived, ProSynth introduces new notions of *"why not"* provenance to compute ω. Intuitively, ω aims to enable certain rules in P^- (i.e., move them to P^+) to force the derivation of t. false negative

As a limiting case, consider the set of all solutions, $P_1^+, P_2^+, \cdots \subseteq P_{all}$, to the synthesis problem (I, T_{exp}, P_{all}), and the induced boolean formula, $\varphi_\star$, whose satisfying assignments correspond exactly to this space of solutions. One way to conceptualize the evolution of φ is as a process of using provenance information to produce successively more precise over-approximations of $\varphi_\star$. As a result, if the synthesis constraint φ ever becomes unsatisfiable, no program satisfies the given specification, and the process terminates without a solution.

2.3 Provenance-Guided Synthesis

Finally, we illustrate the computation of why and why not provenance in selected iterations of ProSynth on our running example. In doing so, we elaborate on the computation of the ψ and ω constraints in Figure 2. We focus on the crucial role played by provenance-guided in accelerating the overall synthesis process. We provide a detailed log of one run of ProSynth on our running example in Appendix B.

2.3.1 Strengthening φ with Why Provenance. The initial value of the synthesis constraint, φ_0 = TRUE, so that in the first iteration, ProSynth starts out with the candidate program $P^+ = P_{all} = \{r_0, r_1, \ldots, r_7\}$. Evaluating this program on the given input relation edge in Figure 1 derives many undesirable tuples in the output relation scc. For each such tuple, ProSynth determines a subset of rules in P^+ whose simultaneous presence resulted in the derivation of this tuple, with the goal of suppressing that combination of rules in future iterations. Off-the-shelf Datalog solvers such as Soufflé [Jordan et al. 2016] come with facilities to efficiently compute these derivation trees, which we call the "*why*" provenance.

For example, consider the undesirable tuple scc(6, 1), one of whose derivation trees is depicted in Figure 3a. ProSynth collects the set of rules used in this tree, $\{r_1, r_2, r_3, r_6, r_7\}$, and updates φ to suppress this combination of rules in future iterations:

$$\varphi_1 = \varphi_0 \wedge \neg(v_1 \wedge v_2 \wedge v_3 \wedge v_6 \wedge v_7).$$

In the second iteration, the SAT solver returns the candidate program $\{r_1, r_4, r_5, r_6\}$, which is a valid solution to φ_1. However, this program also derives undesirable tuples such as scc(3, 5), one of whose derivation trees is depicted in Figure 3b. As before, ProSynth collects the set of rules used in this tree, $\{r_1, r_4\}$, and updates φ:

$$\varphi_2 = \varphi_1 \wedge \neg(v_1 \wedge v_4).$$

Notice that ProSynth has evaluated only two candidate programs so far but has eliminated 34 out of the $2^8 = 256$ possible programs as being unviable (that is, 4 in the first iteration, 32 in the second iteration, and 2 in both iterations).

2.3.2 Strengthening φ with Why Not Provenance. In the fifth iteration, the constraint solver proposes the candidate program $P^+ = \{r_1, r_2\}$, which is consistent with the feedback in iterations 1–4. This program fails to derive many desirable tuples, including t = scc(3, 1). The constraint solver has clearly excluded too many rules from P^+, and at least one of the rules in P^- must necessarily be in any candidate solution. Therefore, a naive approach is to simply update φ as follows:

$$\varphi_5' = \varphi_4 \wedge (v_0 \vee v_3 \vee v_4 \vee v_5 \vee v_6 \vee v_7),$$

While the additional constraint disallows the current candidate program from being generated in future iterations, it does not perform any generalization, and is consequently very weak. In particular, of the 256 programs in the search space, it only disallows the $2^{|P^+|} = 4$ programs that are subsets of P^+. As we will show in Section 5, in practice, synthesis with this approach is slow to converge, and requires 8 iterations on average for our example task.

One of the contributions of this paper is in introducing techniques to perform a tighter analysis of the failure of candidate programs to produce desirable tuples. Conceptually, this corresponds to growing P^+ and shrinking P^- while still ensuring that the desirable tuple t fails to be produced. See Figure 4. As a result, instead of the original constraint,

$$\omega = \bigvee_{r \in P^-} v_r,$$

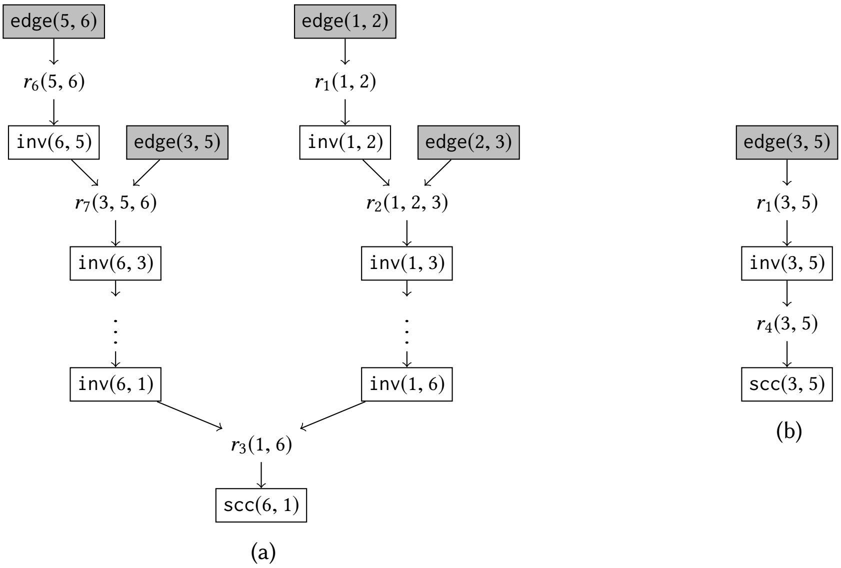


Fig. 3. Derivation trees resulting in the production of scc(6, 1) and scc(3, 5) in the first (Figure 3a) and second (Figure 3b) iterations of ProSynth in our example synthesis task.

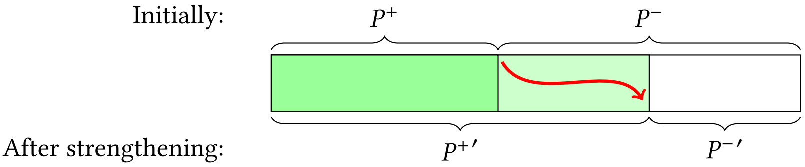


Fig. 4. The original candidate program P^+ failed to derive some desirable tuple t, and we subsequently added candidate rules to P^+ while still preserving the non-production of t. As a result, the set P^- of excluded rules shrinks to a smaller set $P^{-\prime}$, thus strengthening the associated why-not constraint.

we obtain the stronger constraint,

$$\omega' = \bigvee_{r \in P^{-\prime}} v_r,$$

for some (hopefully much smaller) subset, $P^{-\prime} \subseteq P^-$.

Our insight is to view the set of excluded rules, P^- as a "*program*", and the failure to produce t as a "*bug*". We then use the delta-debugging algorithm [Zeller 1999] to derive a minimal subprogram, $P_{min} \subseteq P^-$, which also exhibits this bug, i.e., $t \notin P^{+\prime}(I)$, where $P^{+\prime} = P_{all} \setminus P_{min}$. For the tuple scc(3, 1), by applying this procedure, we obtain $P_{min} = \{r_0, r_3, r_4, r_5\}$, and we therefore update φ as follows:

$$\varphi_5 = \varphi_4 \wedge (v_0 \vee v_3 \vee v_4 \vee v_5).$$

Recall that the naive failure analysis only eliminated $2^{|P^-|} = 4$ candidate programs. In contrast, minimizing the size P^- to P_{min} eliminates $2^{|P_{all} \setminus P_{min}|} = 2^{8-4} = 16$ candidate programs from consideration. In practice, using the strong why-not explanations instead of the naive approach enables ProSynth to converge in just 5 iterations on average.

We could do even better by introducing other forms of why not provenance. We describe one such notion that we call *co-provenance*. While why-not explanations serve to explain the non-derivation of a desirable tuple after the fact, co-provenance serves to proactively determine combinations of rules necessary to avoid it from happening in the first place. For example, in the fourth iteration, the constraint solver proposes $P^+ = \{r_1, r_2, r_5, r_6\}$, which we recall below:

$$\begin{aligned}
r_1 :&\quad \texttt{inv}(x, y) \text{ :- } \texttt{edge}(x, y). \\
r_2 :&\quad \texttt{inv}(x, z) \text{ :- } \texttt{inv}(x, y), \texttt{edge}(y, z). \\
r_5 :&\quad \texttt{scc}(x, y) \text{ :- } \texttt{inv}(y, x). \\
r_6 :&\quad \texttt{inv}(x, y) \text{ :- } \texttt{edge}(y, x).
\end{aligned}$$

This program is clearly incorrect, because it produces several undesirable tuples, as we observe in Appendix B. However, it does produce the desired output tuple $t = \texttt{scc}(3, 1)$. Note however, that in this context, every derivation tree producing t contains an occurence of the rule r_5. This observation allows us to conclude, without any further evaluation, that the candidate program $P^+ \setminus \{r_5\} = \{r_1, r_2, r_6\}$ will not produce t. In other words, if all currently negative rules, $P^- = \{r_0, r_3, r_4, r_7\}$ continue to be excluded, then r_5 must continue to be included, and we could update φ with the following co-provenance constraint:

$$\varphi_4' = \varphi_3 \land ((\neg v_0 \land \neg v_3 \land \neg v_4 \land \neg v_7) \implies v_5),$$

which would subsequently prevent the failure to derive $\texttt{scc}(3, 1)$ in the fifth iteration.

In our experiments, the scc benchmark contains 166 candidate rules. ProSynth finds the desired program in 16 seconds, invoking the SAT solver 81 times and the Datalog solver 844 times. In contrast, a version-space search based system, ALPS [Si et al. 2018], takes 56 seconds and invokes the Datalog solver 47,527 times, reflecting modest ability to generalize from failures. Likewise, a numerical relaxation based system Difflog [Si et al. 2019] takes 47 minutes and invokes the Datalog solver 4,008 times—each invocation of the solver is significantly more expensive because the same set of 166 rules is run in each iteration with different rule weights.

3 THE DATALOG SYNTHESIS PROBLEM

In this section, we formalize the Datalog synthesis problem. We start by briefly reviewing the main ideas underlying Datalog, as presented in [Abiteboul et al. 1994], formulate the rule selection problem, and survey some elementary hardness results.

3.1 Overview of Datalog

We first fix a finite set of input relation names $\mathcal{I}$ and a finite set of output relation names $\mathcal{O}$. Each relation $R \in \mathcal{I} \cup \mathcal{O}$ is a set of tuples of the form $R(c_1, c_2, \dots, c_k)$ of appropriate arity. In the example synthesis task in Section 2, $\mathcal{I} = \{\texttt{edge}\}$ and $\mathcal{O} = \{\texttt{inv}, \texttt{scc}\}$.

Then, we explicitly list the set of tuples I, which populate the input relations, and implicitly define output relations using a finite set of *rules*. Each rule r is a Horn clause of the form:

$$R_h(\boldsymbol{v}_h) \text{ :- } R_1(\boldsymbol{v}_1), R_2(\boldsymbol{v}_2), \dots, R_k(\boldsymbol{v}_k),$$

where the arguments, $\boldsymbol{v}_h$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, …, $\boldsymbol{v}_k$, are vectors of variables of appropriate arity. An example is the rule for transitive closure: $\texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{edge}(y, z)$. The literal to the left of the ":-" operator, $R_h(\boldsymbol{v}_h)$, is called the *head*, and always references an output relation, while the literals on the right hand side, $R_1(\boldsymbol{v}_1)$, $R_2(\boldsymbol{v}_2)$, …, $R_k(\boldsymbol{v}_k)$, form the *rule body*.

Each rule is read from right-to-left as a universally quantified implication: "*For all variable valuations $\boldsymbol{v}$, if each of the tuples $R_1(\boldsymbol{v}_1)$, $R_2(\boldsymbol{v}_2)$, …, and $R_k(\boldsymbol{v}_k)$ is derivable, then so is $R_h(\boldsymbol{v}_h)$* ". A Datalog program is a finite set of rules $P = \{r_1, r_2, \dots, r_n\}$. Multiple equivalent formalizations have

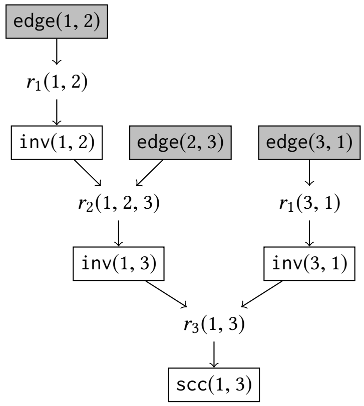


Fig. 5. A portion of the derivation graph of the output tuple scc(1, 3), obtained by applying the rules r_1, r_2 and r_3 to the input graph of Figure 1. The nodes corresponding to rule instances, such as $r_2(1, 2, 3)$, also present the values with which the variables were instantiated. With this particular choice of rules, the invented predicate inv represents the familiar notion of reachability in a graph: inv(x, y) is derivable iff the vertex y is reachable from the vertex x.

been proposed for their semantics: informally, one starts with the input tuples I, and repeatedly instantiates the variables of each rule to derive new output tuples, until no further conclusions can be reached. At fixpoint, we obtain a derivation graph containing the input and output tuples, and the rule instantiations which connect them, such as that shown in Figure 5. We will write $T_{out} = P(I)$ for the set of output tuples produced by a Datalog program operating on a set of input tuples I. There is a large body of research on efficiently evaluating Datalog programs and several high-quality commercial and open-source Datalog solvers are available [Aref et al. 2015; Jordan et al. 2016; Whaley and Lam 2004].

3.2 Synthesis as Rule Selection

In this paper, we follow the popular syntax-guided approach to synthesizing Datalog programs. There are two principal ingredients of a syntax-guided synthesis (SyGuS) problem [Alur et al. 2013]: (*a*) an input-output specification which constrains the *behavior* of the target program, and (*b*) a grammar which specifies the *syntactic shape* of the target program.

As usual, the user provides a set of input tuples I. In the case of the output, for the sake of generality, we slightly depart from the example of Section 2, and instead of a single output specification T_{exp}, use two parameter sets T^+_{exp} and T^-_{exp}, corresponding to an explicit, non-exhaustive labelling of desired and undesired output tuples respectively. The goal of the synthesizer is to find a Datalog program P which transforms I into a set of output tuples $P(I)$ such that: (*a*) all desirable tuples are successfully produced, i.e., $T^+_{exp} \subseteq P(I)$, and (*b*) no undesirable tuple is produced, i.e., $T^-_{exp} \cap P(I) = \emptyset$.

We emphasize that the labeling of output tuples into T^+_{exp} and T^-_{exp} need not be exhaustive: the user may choose to identify as many or as few output tuples as they desire. A prominent setting in which this flexibility is important is in the case of *invented predicates*, such as the relation inv of Section 2, where none of the tuples of the particular output relation are labeled. However, this under-constrained nature of the problem specification also greatly increases the difficulty of program synthesis, as it is now not possible to consider an individual rule and determine whether it is the cause of undesirable behavior in the candidate program. The proof of Theorem 3.2 in [Si et al. 2019] crucially exploits this observation.

The second component of syntax-guided synthesis is a grammar which constrains the possible target programs. Various forms of syntax guidance have been previously used in the literature on inductive logic programming, for example as *metarules* [Muggleton et al. 2015; Si et al. 2018] and as restrictions on the numbers and forms of possible solution rules [Quinlan and Cameron-Jones 1995; Zeng et al. 2014]. In this context, syntax guidance may be viewed as a form of *inductive bias* [Kitzelmann 2010], both to enable efficient convergence to the target program, and to prevent overfitting. In this paper, we follow [Si et al. 2019] in uniformly capturing all these forms of syntactic bias in the form of rule selection: we assume that the user provides a large soup of *candidate rules*, P_{all}, and the goal of the synthesizer is to find a concrete Datalog program $P \subseteq P_{all}$ with the desired input-output behavior.

In summary, the central technical problem of this paper is the following:

Problem 3.1 (Rule Selection). *Given finite sets of input tuples I, desirable output tuples T^+_{exp}, undesirable output tuples T^-_{exp}, and candidate rules P_{all}, find a Datalog program $P \subseteq P_{all}$, such that $T^+_{exp} \subseteq P(I)$ and $T^-_{exp} \cap P(I) = \emptyset$.*

The key to solving the rule selection problem is to identify rules which cause undesirable behavior. Unfortunately, the correctness of an individual candidate rule $r \in P \subseteq P_{all}$ is determined not just by r but also by the *other rules* r' present in the candidate program P. We have seen an example of this behavior in the example of Section 2, where both $P = \{r_1, r_2, r_3\}$ and $P' = \{r_3, r_6, r_7\}$ are valid solutions, but $P \cup P'$ is not a valid solution. The following hardness result follows from a straightforward encoding of the satisfiability of a 3-CNF formula:

Theorem 3.2 ([Si et al. 2019]). *Deciding whether an instance of the rule selection problem, $(I, T^+_{exp}, T^-_{exp}, P_{all})$, admits a solution is NP-hard.*

3.3 Generating Candidate Rules

While the ProSynth framework is agnostic of the choice of candidate rules P_{all}, in our experiments in Section 5, we consider two approaches to generate P_{all}, which we will now describe.

Generating candidate rules by augmentation. A common approach to generating candidate rules is by instantiating *meta-rules* [Muggleton et al. 2015]. A meta-rule is a higher-order rule containing named holes in place of the concrete relation names found in a traditional Datalog rule. The following expression is an example of a meta-rule:

$$P_0(x, z) \text{ :- } P_1(x, y), P_2(y, z). \quad (1)$$

Here P_0, P_1 and P_2 are higher-order named holes, also called *meta-variables*, and can be consistently replaced with concrete relation names such as edge, inv or scc to obtain traditional (first-order) rules. Instantiating the meta-rule in this example will yield a large set of candidate rules, including "inv(x, z) :- edge(x, y), edge(y, z)", "scc(x, z) :- scc(x, y), inv(y, z)", etc.

[Muggleton et al. 2015] require the user to explicitly provide these meta-rules. In contrast, [Si et al. 2018] proposed a technique named *augmentation* to automatically generate these meta-rules: they noticed that certain patterns, such as the variable chain in meta-rule (1) are common in a variety of Datalog programs. The idea is to begin with a small set of manually curated meta-rules, and repeatedly apply edits to the existing meta-rules to generate new meta-rules. Possible edits include inserting, deleting, or renaming a variable from a literal, and changing the names of meta-variables. Thus, for example, meta-rule (1) may be modified to obtain:

$$P_0(x, z) \text{ :- } P_1(x, y), P_2(y, y), \text{ and}$$

$$P_0(x, y, z) \text{ :- } P_1(x, y), P_2(y, z).$$

We remark that, in the context of the SCC synthesis example of Section 2, the second meta-rule above cannot generate any candidate rules because there are no relations of arity 3. Notice, however, that one can generate any candidate Datalog rule of a given width with a sufficiently large number of augmentation steps. In their experiments, [Si et al. 2018] observed that by augmenting the chain meta-rules to a depth of 5, and by then exhaustively instantiating them, they were able to generate most candidate rules in their benchmark set. In our experiments, we reuse the same set of meta-rules from their publicly available artifact.

Generating candidate rules by enumerating literals. A second way to generate candidate rules is by enumerating all sequences of literals of a given width. Let k be the maximum chosen size of the rule body. The idea is to enumerate, for each output relation, all clauses of width less than or equal to k, and with all possible combinations of variable assignments. We then perform a sequence of normalization passes, to eliminate ill-formed and duplicate rules: we ensure that the types of variable arguments match the relation schemas and normalize variable names to identify equivalent rules. This mechanism is a convenient way to generate large numbers of candidate rules, and we use it (with $k = 3$) in our experiments in Section 5.4 to study the scalability of ProSynth with respect to varying sizes of P_{all}.

4 PROVENANCE-GUIDED SYNTHESIS FRAMEWORK

Algorithm 1 formalizes the interaction process described in Figure 2. Starting with $\varphi := \text{TRUE}$, the algorithm iteratively strengthens the synthesis constraint to disallow non-solutions. In each iteration, it first finds a satisfying assignment of the formula φ by querying a SAT solver. This determines the candidate program P^+, such that a rule r is in P^+ iff the variable v_r is set to true in the satisfying assignment returned by the SAT solver (step 2a). A Datalog solver evaluates the candidate program P^+ for the given set of input tuples I to obtain the set of output tuples $T_{out} = P^+(I)$ (step 2b). This output T_{out} of the candidate program P^+ may or may not conform to the output specification (T^+_{exp}, T^-_{exp}). If it satisfies the output specification (step 2c), then the algorithm returns the candidate program P^+ as a valid solution to the synthesis problem. Otherwise, the iteration analyzes the reasons for the failure of the candidate program and strengthens the synthesis constraint φ. As discussed in Section 2, this failure analysis has three steps:

(1) $t \in T^-_{exp} \cap T_{out}$ is an undesirable tuple for which we expand φ with a *why* provenance constraint (step 2d),
(2) $t \in T^+_{exp} \setminus T_{out}$ is an unproduced desirable tuple for which we expand φ with a *why not* provenance constraint (step 2e),
(3) $t \in T^+_{exp} \cap T_{out}$ is a produced desirable for which we expand φ with a *co-provenance* constraint (step 2f).

If, at any point, the synthesis constraint φ becomes undecidable, then the algorithm concludes that the problem does not admit any solutions and terminates (steps 2 and 3).

We note that the overall approach is a synthesis framework based on provenance rather than a single monolithic algorithm, and enables many different optimizations and diverse notions of provenance. In particular, step 2f is optional, and the why not provenance of step 2e may be optionally strengthened using the delta-debugging procedure of Section 4.2.2.

Throughout the ensuing discussion, we will freely conflate assignments to the boolean variables, M, and candidate programs P^+. To emphasize the construction of one object from the other, we will occasionally write M_{P^+} and P^+_M: Given a boolean assignment M, the corresponding candidate program P^+_M may be computed using Equation 2 of Algorithm 1. The construction in the reverse

Algorithm 1 ProSynth$(I, T^+_{exp}, T^-_{exp}, P_{all})$. Given an instance of the rule selection problem, returns a satisfying solution $P \subseteq P_{all}$, or none if no solution exists.

```
(1) Associate each rule $r \in P_{all}$ with a Boolean variable $v_r$. The synthesis constraint $\varphi$ ranges
    over these variables. Initialize $\varphi := \textsc{true}$.
(2) Repeat until $\varphi$ is no longer satisfiable:
    (a) Let $M$ be the satisfying assignment returned by the SAT solver. Define rulesets
            $P^+ = \{r \in P_{all} \mid M(v_r) = \textsc{true}\}, \text{ and}$    (2)
            $P^- = \{r \in P_{all} \mid M(v_r) = \textsc{false}\}.$    (3)
    (b) Compute $T_{out} = P^+(I)$.
    (c) If $T^+_{exp} \subseteq T_{out}$ and $T^-_{exp} \cap T_{out} = \emptyset$, then return $P^+$.
    (d) For every produced undesirable tuple $t \in T^-_{exp} \cap T_{out}$: compute $\psi = \text{prov}(t, P^+)$, and update
        the synthesis constraint: $\varphi := \varphi \wedge \neg\psi$.
    (e) For every unproduced desirable tuple $t \in T^+_{exp} \setminus T_{out}$: compute $\omega = \overline{\text{prov}}(t, P^-)$, and update
        the synthesis constraint: $\varphi := \varphi \wedge \omega$.
    (f) (Optionally) For every produced desirable tuple $t \in T^+_{exp} \cap T_{out}$: compute $\sigma = \text{keep}(t, P^+, P^-)$,
        and update the synthesis constraint: $\varphi := \varphi \wedge \sigma$.
(3) Return none.
```

direction is also straightforward:

$$M_{P^+}(v_r) = \textsc{true} \text{ iff } r \in P^+. \tag{4}$$

In the rest of this section, we elaborate on Algorithm 1 by describing the computation of the $\text{prov}(t, P^+)$, $\overline{\text{prov}}(t, P^-)$, and $\text{keep}(t, P^+, P^-)$ constraints. We begin by establishing its correctness.

Theorem 4.1. *Given an instance of the rule selection problem, $Q = (I, T^+_{exp}, T^-_{exp}, P_{all})$, Algorithm 1 returns a valid solution $P \subseteq P_{all}$ iff the problem admits a solution and returns* none *otherwise.*

Proof. First, notice that the algorithm only returns a solution P in step 2c. Here, it is clearly the case that $P \subseteq P_{all}$ and that it is a valid solution to the problem instance Q.

In Lemmas 4.2, 4.3, 4.4, and 4.5, we show that any solution P_v to the rule selection problem Q also satisfies the why, why-not, and co-provenance constraints, $\neg\psi$, ω, and σ. From this, we may establish the invariant that each solution P_v to the rule selection problem is also always a satisfying assignment to the synthesis constraint φ. It subsequently follows that if ProSynth returns none, the problem is unsatisfiable.

Finally, let φ_k be the value of the synthesis constraint after k iterations of the loop, and let M_k be the satisfying assignment to φ_k chosen by the SAT solver. Notice that if M_k does not identify a solution to the synthesis problem Q, then at least one of the steps 2d or 2e triggers, so that M_k is no longer a satisfying assignment to the subsequent synthesis constraint φ_{k+1}. Therefore, the number of satisfying assignments to φ strictly decreases in each loop iteration, so that the algorithm is guaranteed to eventually terminate. This completes the proof. □

The above theorem relies on the invariant that all solutions of the rule selection problem Q are always satisfying assignments to varphi. Notice that the algorithm also permits *fortunate termination*: i.e., even if φ is satisfied by non-solutions, the SAT solver may non-deterministically select a satisfying assignment which solves the problem.

4.1 Why-Provenance: Blocking of Undesirable Tuples

A candidate program $P^+ \subseteq P_{all}$ may erroneously derive some undesirable tuples $t \in T^-_{exp}$ in its output set T_{out}. As examples, recall the first two example iterations in Section 2, where the tuples $\texttt{scc}(6, 1)$ and $\texttt{scc}(3, 5)$ were produced. Also, recall that each tuple produced by a Datalog program is associated with a derivation tree, of which we presented examples in Figures 3a, 3b and 5.

In particular, notice that the derivation tree for $\texttt{scc}(6, 1)$ contains exactly the set of rules $\{r_1, r_2, r_3, r_6, r_7\}$. Therefore, any candidate program P^+ such that $\{r_1, r_2, r_3, r_6, r_7\} \subseteq P^+$ will continue to derive tuple $\texttt{scc}(6, 1)$. Similarly, since the derivation tree for $\texttt{scc}(3, 5)$ contains occurrences of the rules $\{r_1, r_4\}$, any candidate program P^+ such that $P^+ \subseteq \{r_1, r_4\}$ will also derive $\texttt{scc}(3, 5)$. Since neither of these tuples is desirable, we add the constraints $\neg(v_1 \wedge v_2 \wedge v_3 \wedge v_6 \wedge v_7)$ and $\neg(v_1 \wedge v_4)$ to φ_1 and φ_2 respectively to block these derivations.

When queried with an output tuple t and a set of rules P^+, the Datalog solver constructs a derivation tree τ which produces t and emits the set of rules which appear in τ. We refer to this as $\text{prov}(t, P^+)$. Note that t may be the result of multiple (and possibly even infinitely many) distinct derivation trees. Therefore, $\text{prov}(t, P^+)$ is not uniquely defined but is rather the result of a non-deterministic computation. By abuse of notation, by $\text{prov}(t, P^+)$, we will also refer to the conjunction of all rule variables appearing in the set, so that $\text{prov}(\texttt{scc}(6, 1), P_{all}) = v_1 \wedge v_2 \wedge v_3 \wedge v_6 \wedge v_7$. The following lemma formalizes our intuition:

Lemma 4.2. *For each pair of candidate programs $P, P' \subseteq P_{all}$, and for each tuple $t \in P(I)$, if P' satisfies* $\text{prov}(t, P)$*, then $t \in P'(I)$.*

Provenance instrumentation is available in the Soufflé Datalog solver [Zhao et al. 2019]. To compute the provenance of a tuple, Soufflé generates proof trees using a lazy two-phase approach. During the evaluation phase of a Datalog program, Soufflé stores two *proof annotations* for each tuple, corresponding to the rule which generates that tuple, and the minimum height of its proof trees. In the second phase, provenance may be queried, and Soufflé reconstructs the proof tree of smallest height using the proof annotations as constraints while searching the database of computed tuples. This lazy evaluation approach minimizes the overhead required for Datalog evaluation, while also maintaining an efficient provenance query mechanism.

4.2 Why-Not Provenance: Enabling the Production of Desirable Tuples

Standard models of provenance capture precisely the reasons for the production of a particular undesirable tuple. On the other hand, candidate programs might also not derive some desirable output tuples, in which case the synthesis constraint again needs to be strengthened to disallow such programs. Note that the lack of existing derivation trees makes this a fundamentally difficult problem. We introduce two versions of the why not constraint $\overline{\text{prov}}(t, P^-)$. The first form, $\overline{\text{prov}}_S(t, P^-)$, is a naive constraint which only takes all missing rules in P^- into account. The second version $\overline{\text{prov}}_\Delta(t, P^-)$ performs a more elaborate reasoning process that significantly generalizes from the present failure.

4.2.1 *Naive Approach,* $\overline{\text{prov}}_S(t, P^-)$.

Recall the fifth iteration of our example problem in Section 2. In this situation, $P^+ = \{r_1, r_2\}$, so that $P^- = \{r_0, r_3, r_4, r_5, r_6, r_7\}$. The candidate program P^+ did not derive the tuple $t = \texttt{scc}(3, 1)$. Clearly, the constraint solver has excluded too many rules from P^+, and omitting these rules resulted in the failure to derive t. In particular, any viable solution must contain at least one of the rules from P^-, so a naive approach to block this failure in future is to assert the constraint $\omega = v_0 \vee v_3 \vee v_4 \vee v_5 \vee v_6 \vee v_7$.

More generally, if $t \in T^+_{exp} \setminus P^+(I)$ is a desirable-but-unproduced tuple, and $P^- = P_{all} \setminus P^+$, then any candidate solution must satisfy the following constraint:

$$\overline{\text{prov}}_S(t, P^-) = \bigvee_{r \in P^-} v_r. \tag{5}$$

We can then show that:

Lemma 4.3. *Pick an arbitrary pair of candidate programs, $P, P' \subseteq P_{all}$, and a tuple $t \notin P(I)$. Let $P^- = P_{all} \setminus P$. If P' does not satisfy $\overline{\text{prov}}_S(t, P^-)$, then $t \notin P'(I)$.*

On the other hand, observe that we have not performed any generalization: $\overline{\text{prov}}(t, P^-)$ only excludes $2^{|P^+|}$ programs from consideration, i.e., the current candidate program and all its subsets. Unsurprisingly, this approach does not scale well to large benchmarks, and exhibits a large variance in our experiments in Section 5.

4.2.2 Failure Analysis Using Delta Debugging, $\overline{\text{prov}}_\Delta(t, P^-)$. One technical insight of this paper is that techniques from automatic program debugging can be used to strengthen why-not constraints. The idea is to view the set of excluded rules P^- as a *program*, and the non-production of t as a *bug*. We can then use the algorithm for delta debugging [Zeller 1999] to shrink P^- to a smaller set which still fails to produce t. The resulting constraint, $\overline{\text{prov}}_\Delta(t, P^-)$, is therefore shorter than $|P^-|$ and generalizes to eliminate many other candidate programs. We formally describe this procedure in Algorithm 2.

The algorithm partitions the set P^- into approximately equal-sized subsets (step 3a). For each subset Δ_i, the algorithm checks if either Δ_i (step 3c) or its complement, ∇_i (step 3d) is buggy. In either of these cases, we focus on the smaller program whose bugginess we have just witnessed. Otherwise, the algorithm proceeds to decompose P^- into smaller partitions, ultimately finding a minimal buggy subset.

Algorithm 2 $\overline{\text{prov}}_\Delta(t, P^-)$. Given an unproduced set of tuples t, and a set of excluded rules of P^-, produces a smaller set of excluded rules which still fails to derive t.

```
(1) Let $d$ be an integer variable denoting the number of partitions into which $P^-$ is split. Initialize
    $d := 2$.
(2) For a set of rules $P' \subseteq P^-$, say that it is buggy if $(P_{all} \setminus P')$ fails to derive $t$.
(3) While $d \leq |P^-|$:
  (a) Partition $P^-$ into $d$ subsets, $\Delta_1, \Delta_2, \ldots, \Delta_d$ of roughly equal size.
  (b) For each $i$, define $\nabla_i = P^- \setminus \Delta_i$.
  (c) If there exists $i$ such that $\Delta_i$ is buggy, update $P^- := \Delta_i$ and $d := 2$.
  (d) Otherwise, if there exists $i$ such that $\nabla_i$ is buggy, then update $P^- := \nabla_i$ and $d := d - 1$.
  (e) Otherwise, update $d := 2d$.
(4) Return $P^-$.
```

The following result follows along the same lines as [Zeller 1999].

Lemma 4.4. *Let $P^+ \subseteq P_{all}$ fail to derive some tuple t, let $P^- = P_{all} \setminus P^+$ be the set of excluded rules, and let $P^-_\Delta = \overline{\text{prov}}_\Delta(t, P^-)$ be the strengthened why-not constraint. Then:*

(1) $P^-_\Delta \subseteq P^-$, and
(2) $P^+_\Delta = P_{all} \setminus P^-_\Delta$ also fails to derive t, and
(3) for each rule $r \in P^-_\Delta$, $P^+_\Delta \cup \{r\}$ will derive t.

The final constraint in the above theorem indicates *1-minimality*, i.e., the idea that removing any individual rule from P^-_Δ will allow the resulting program to derive t, and thus make the constraint too strong to be valid.

Once again, recall the fifth iteration of our example in Section 2, where we had an unproduced desirable tuple $t = \texttt{scc}(3, 1)$. The complement of the candidate program is $P^- = \{r_0, r_3, r_4, r_5, r_6, r_7\}$. Using the delta-debugging procedure, we repeatedly test subsets of P^- to ultimately discover that the subset $P_{min} = \{r_0, r_3, r_4, r_5\}$ also exhibits the "*bug*" of not deriving t. Note that this is to be expected: the remaining rules,

$$\begin{aligned} r_1 : &\quad \texttt{inv}(x, y) \text{ :- } \texttt{edge}(x, y). \\ r_2 : &\quad \texttt{inv}(x, z) \text{ :- } \texttt{inv}(x, y), \texttt{edge}(y, z). \\ r_6 : &\quad \texttt{inv}(x, y) \text{ :- } \texttt{edge}(y, x). \\ r_7 : &\quad \texttt{inv}(x, z) \text{ :- } \texttt{inv}(x, y), \texttt{edge}(z, y). \end{aligned}$$

only derive elements of the invented predicate, and the candidate program cannot possibly be correct if there is no rule that produces tuples which inhabit the scc relation. Hence, for $t = \texttt{scc}(3, 1)$, the why not provenance constraint, $\overline{\text{prov}}_\Delta(t, P^-) = v_0 \vee v_3 \vee v_4 \vee v_5$. As we discussed in Section 2, while the naive constraint $\overline{\text{prov}}_S(t, P^-)$ eliminates $2^{|P^+|} = 4$ candidate programs, the strengthened form, $\overline{\text{prov}}_\Delta(t, P^-)$ eliminates $2^{|P_{all} \setminus P_{min}|} = 16$ candidate programs from consideration.

4.3 Co-Provenance: Keeping Produced Desirable Tuples

Consider the fourth iteration of the algorithm in Section 2. In this situation, $P^+ = \{r_1, r_2, r_5, r_6\}$. This candidate program does successfully produce the tuple $t = \texttt{scc}(3, 1)$. In fact, because of the loop in the underlying graph, there are infinitely many derivation trees which produce t. Note, however, that three of the included rules, r_1, r_2 and r_6, only produce elements of the intermediate predicate inv. Therefore, all derivation trees which produce t *must* ultimately involve an occurrence of the remaining rule r_5.

The notion of *co-provenance* captures this idea of a rule being essential to the production of a tuple. Formally, the co-provenance of a tuple t, $\text{coprov}(t, P^+)$ is the set of all rules which occur in every derivation tree producing t:

$$\text{coprov}(t, P^+) = \{r \in P^+ \mid \forall \text{ derivation trees } \tau \text{ which produce } t, r \in \tau\}. \tag{6}$$

While $\text{prov}(t, P^+)$ refers to one non-deterministically chosen derivation tree, the quantity defined above simultaneously refers to *all* derivation trees of t, thus making *co*-provenance a dual to the traditional notion of provenance.

Furthermore, unlike $\text{prov}(t, P^+)$ which we defined in Section 4.1 as the result of a non-deterministic computation, Equation 6 always uniquely defines the quantity $\text{coprov}(t, P^+)$. We discuss two pathological cases to clarify the concept:

(1) Consider a tuple $t \in P^+(I)$ which has two derivation trees, τ_1 and τ_2, where τ_1 and τ_2 do not contain any rules in common. In this situation, no single rule is necessary for the production of t, and therefore, $\text{coprov}(t, P^+) = \emptyset$.
(2) Consider a tuple $t \notin P^+(I)$ which is not derived by the candidate program P^+. In this case, there are no derivation trees which produce t, and therefore, each rule r vacuously satisfies the condition of Equation 6. Therefore, $\text{coprov}(t, P^+) = P^+$.

Lastly, note that Equation 6 merely *defines* the concept and does not present an algorithm to compute $\text{coprov}(t, P^+)$. We will present a technique to compute the quantity in the second part of this section.

4.3.1 Using $\mathrm{coprov}(t, P^+)$ *to define* $\mathrm{keep}(t, P^+, P^-)$. Our main idea is that in the context where no additional derivation trees are present, the rules present in $\mathrm{coprov}(t, P^+)$ *necessarily* have to be present to derive t. In particular, since $r_5 \in \mathrm{coprov}(\mathtt{scc}(3, 1), P^+)$, for any candidate program P', if $P' \cap P^- = \emptyset$ and $r_5 \notin P'$, then $\mathtt{scc}(3, 1) \notin P'(I)$.

We therefore define the constraint keep that protects the rules necessary for the production of desirable tuples as follows,

$$\mathrm{keep}(t, P^+, P^-) = \bigwedge\nolimits_{r \in P^-} \neg v_r \implies \bigwedge\nolimits_{r \in \mathrm{coprov}(t, P^+)} v_r. \tag{7}$$

The keep constraint is an implication based on the given set P^-. It permits us to remove further rules from P^+ that may otherwise produce undesirable tuples, i.e., rules in the set $P^+ \setminus \mathrm{coprov}(t, P^+)$.
and observe that the following lemma follows:

Lemma 4.5. *Pick candidate programs P^+ and P', and let $P^- = P_{all} \setminus P^+$. For each tuple t, if P' does not satisfy* $\mathrm{keep}(t, P^+, P^-)$*, then $t \notin P'(I)$.*

Proof. Assume otherwise, so that $t \in P'(I)$.

If P' does not satisfy $\mathrm{keep}(t, P^+)$, it has to be the case that it satisfies $\bigwedge_{r \in P^-} \neg v_r$, but does not satisfy $\bigwedge_{r \in \mathrm{coprov}(t, P^+)} v_r$. We can therefore assert that $P' \subseteq P$, and also that there exists a rule r such that $r \in \mathrm{coprov}(t, P^+)$ such that $r \notin P'$.

Now pick a derivation tree τ which derives t in $P'(I)$, and observe that this derivation tree could also have been realized in $P^+(I)$. Since this tree τ does not contain r, it contradicts the assumption that $r \in \mathrm{coprov}(t, P^+)$. □

4.3.2 Computing $\mathrm{coprov}(t, P^+)$. The central difficulty with computing $\mathrm{coprov}(t, P^+)$ is that it references *all* derivation trees of a particular conclusion, whereas traditional Datalog solvers are best suited to discovering facts which hold on some derivation tree. Our insight is to compute the complement of the co-provenance set. In other words, a rule r cannot belong to $\mathrm{coprov}(t, P^+)$ iff there is at least one derivation tree of t in which r does not occur.

For example, consider the two derivation trees of $t = \mathtt{scc}(3, 1)$ shown in Figure 6. Because the rules r_1 and r_2 do not occur in τ_1, we can conclude that $r_1, r_2 \notin \mathrm{coprov}(t, P^+)$. Similarly, since r_6 does not appear in τ_2, we can conclude that $r_6 \notin \mathrm{coprov}(t, P^+)$. After maximally deriving all "*not-in-the-co-provenance*" facts, we are allowed to conclude that $\mathrm{coprov}(t, P^+) = \{r_5\}$.

Inspired by this reasoning, instead of the co-provenance, which has a universal quantifier in its definition, we compute the following existential quantity:

$$\overline{\mathrm{coprov}}(t, P^+) = \{r \in P^+ \mid \exists \text{ a derivation tree } \tau \text{ s.t. } r \notin t\}. \tag{8}$$

From this definition, it follows that $\mathrm{coprov}(t, P^+) = P^+ \setminus \overline{\mathrm{coprov}}(t, P^+)$.

The key to computing $\overline{\mathrm{coprov}}(t, P^+)$ is to instrument every k-place relation $R(\boldsymbol{v})$ of P^+ with a $(k + 1)$-place *shadow relation* $R^{\exists\neg}(\boldsymbol{v}, r)$ such that $R^{\exists\neg}(\boldsymbol{v}, r)$ is derivable iff $R(\boldsymbol{v})$ is derivable without using rule r. First, every rule r' of the form:

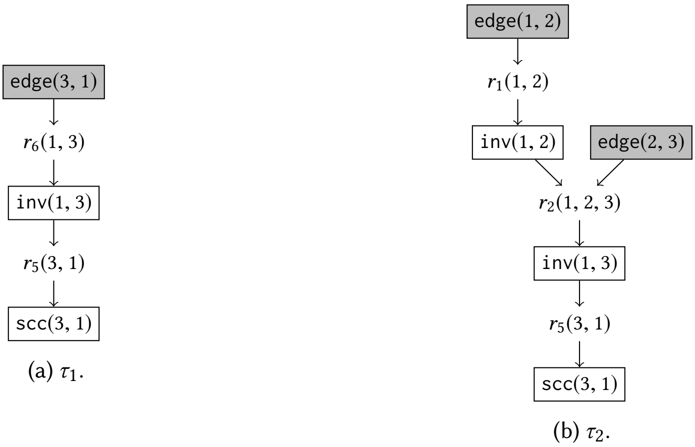
$$r' : \quad R_h(\boldsymbol{v}_h) \text{ :- } R_1(\boldsymbol{v}_1), R_2(\boldsymbol{v}_2), \ldots, R_k(\boldsymbol{v}_k),$$

is instrumented with a shadow rule of the form:

$$R_h^{\exists\neg}(\boldsymbol{v}_h, r) \text{ :- } R_1^{\exists\neg}(\boldsymbol{v}_1, r), R_2^{\exists\neg}(\boldsymbol{v}_2, r), \ldots, R_k^{\exists\neg}(\boldsymbol{v}_k, r), r \neq r'.$$

Informally, if $r \neq r'$ and if each of the hypotheses tuples, $R_1(\boldsymbol{v}_1)$, $R_2(\boldsymbol{v}_2)$, ..., $R_k(\boldsymbol{v}_k)$, are derivable without using rule r, then $R_h(\boldsymbol{v}_h)$ is also derivable without using r. Second, since input tuples are derivable without depending on any rules, each input relation is simply instrumented with a rule of the form:

$$R^{\exists\neg}(\boldsymbol{v}, r) \text{ :- } R(\boldsymbol{v}), r \in P^+.$$



(a) τ_1.

(b) τ_2.

Fig. 6. Two derivation trees of t = scc(3, 1) from the fourth iteration of the example in Section 2. Here, the candidate program, $P^+ = \{r_1, r_2, r_5, r_6\}$. Notice that r_1 and r_2 do not appear in the first tree, τ_1. Therefore, $r_1, r_2 \notin \mathrm{coprov}(t, P^+)$. Similarly, rule r_6 does not occur in τ_2. Therefore, $r_6 \notin \mathrm{coprov}(t, P^+)$.

It can then be shown that, for each relation $R(\boldsymbol{v})$, $R^{\exists\neg}(\boldsymbol{v}, r)$ is derivable iff $r \in \overline{\mathrm{coprov}}(t, P^+)$. In our example, with $P^+ = \{r_1, r_2, r_5, r_6\}$, this results in the following instrumented program:

$$\begin{aligned}
& \mathsf{edge}^{\exists\neg}(x, y, r) \text{ :- } \mathsf{edge}(x, y), r \in P^+. \\
r_1^{\exists\neg} : \quad & \mathtt{inv}^{\exists\neg}(x, y, r) \text{ :- } \mathsf{edge}^{\exists\neg}(x, y, r), r \neq r_1. \\
r_2^{\exists\neg} : \quad & \mathtt{inv}^{\exists\neg}(x, z, r) \text{ :- } \mathtt{inv}^{\exists\neg}(x, y, r), \mathsf{edge}^{\exists\neg}(y, z, r), r \neq r_2. \\
r_5^{\exists\neg} : \quad & \mathtt{scc}^{\exists\neg}(x, y, r) \text{ :- } \mathtt{inv}^{\exists\neg}(y, x, r), r \neq r_5. \\
r_6^{\exists\neg} : \quad & \mathtt{inv}^{\exists\neg}(x, y, r) \text{ :- } \mathsf{edge}^{\exists\neg}(y, x, r), r \neq r_6.
\end{aligned}$$

5 EMPIRICAL EVALUATION

We have implemented ProSynth in Python. It uses Soufflé [Jordan et al. 2016] as the underlying Datalog solver and Z3 [de Moura and Bjørner 2008] as the SAT solver. In this section, we evaluate ProSynth to answer the following questions:

Q1: How effective is ProSynth on synthesis tasks from different domains in terms of synthesis time and learnability as compared to state-of-the-art approaches?
Q2: How variable is the running time of ProSynth across different runs and how does the variability compare to that of existing approaches?
Q3: How does ProSynth scale with respect to the number and nature of candidate rules?
Q4: What is the impact of different provenance-based optimizations in ProSynth?

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

Table 1. Benchmark characteristics.

Benchmark	Brief description	#Relations	#Rules	#Inv. Preds.	Recursive?
Knowledge Discovery					
abduce	find grandparent of given parent [Muggleton 1995]	4	3	0	
animals	distinguishing animal classes [Muggleton 1995]	13	4	0	
buildwall	learn a stable wall strategy [Muggleton et al. 2015]	5	4	1	✓
cliquer	compute 2-paths and SCCs in directed graph	4	4	1	
inflammation	bladder inflammation diagnosis [Czerniak and Zarzycki 2003]	7	2	0	
nearlyscc	all-pairs reachability in directed graph in either direction	2	4	1	
path	all-pairs reachability in directed graph	2	2	0	✓
rsg	reverse-same-generation in family tree [Abiteboul et al. 1994]	4	2	0	✓
samegen	same generation in family tree [Abiteboul et al. 1994]	3	3	0	✓
scc	compute SCCs in directed graph	3	3	1	✓
ship	pairing products with customers by city and product name	4	1	0	
small	find ancestor in a family tree [Muggleton et al. 2015]	4	4	1	
traffic	crashes at an intersection	5	3	1	
unionfind	checking if elements in same set after union operations	4	4	0	
Program Analysis					
1-call-site	1-call-site pointer analysis for Java [Whaley and Lam 2004]	9	4	0	✓
1-object	1-object-sensitive pointer analysis [Milanova et al. 2002]	11	4	0	✓
1-object-type	1-type-1-object sensitive analysis [Smaragdakis et al. 2011]	13	5	0	✓
1-type	1-type-sensitive pointer analysis [Smaragdakis et al. 2011]	12	4	0	✓
2-call-site	2-call-site pointer analysis for Java [Whaley and Lam 2004]	9	4	0	✓
andersen	inclusion-based pointer analysis for C [Andersen 1994]	5	4	0	✓
downcast	downcast safety checker for Java [Si et al. 2018]	9	4	0	
escape	escape analysis for Java [Si et al. 2018]	10	6	0	✓
modref	mod-ref analysis for Java [Si et al. 2018]	13	10	0	✓
polysite	polymorphic call-site inference for Java [Si et al. 2018]	6	3	0	
rvcheck	return-value-checker in APISan [Yun et al. 2016]	5	5	4	
Relational Queries					
sql 1 ∼ 15	15 SQL queries [Wang et al. 2017]	≤ 7	≤ 4	≤ 3	

5.1 Benchmark Suite

We collected 40 synthesis tasks from three different application domains: (*a*) knowledge discovery, (*b*) program analysis, and (*c*) relational queries. Table 1 presents characteristics of these benchmarks, including the number of input-output relations, the number of rules in the smallest desired program, the number of invented predicates needed, and whether the desired program is recursive or not.

Knowledge discovery. These benchmarks comprise 14 tasks of synthesizing Datalog programs frequently used in the artificial intelligence and database literature.

Program analysis. These benchmarks comprise 11 tasks of synthesizing static analysis algorithms for imperative and object-oriented programs.

Relational queries. These benchmarks comprise 15 tasks from StackOverflow posts and textbook examples. They involve synthesizing SQL queries that can be expressed in Datalog.

5.2 Performance and Learnability Results

The central question we wish to investigate is whether ProSynth can effectively learn non-trivial Datalog programs, especially in comparison to other state-of-the-art approaches, including ALPS [Si et al. 2018] and Difflog [Si et al. 2019]. Previous work demonstrates that these two publicly available tools out-perform prior approaches such as Metagol [Cropper and Muggleton 2015] and Zaatar [Albarghouthi et al. 2017]. We present the results of this evaluation in Table 2. Across all benchmarks, ProSynth never times out, takes less than 10 seconds on average to synthesize the target program, and less than 1 second for 28 of them. In comparison, both ALPS and Difflog run out of time on 6 and 3 benchmarks, respectively. Furthermore, ProSynth demonstrates faster performance than both competing solvers in all but 5 of the tasks.

For example, consider the scc benchmark. ProSynth makes 81 calls to the Z3 solver, and evaluates 844 different candidate solutions using Soufflé, and correctly synthesizes the target program in 16 seconds. In comparison, ALPS evaluates 47,527 candidate programs, and Difflog evaluates 4,008 candidate programs before reaching a solution.

Also, observe that ProSynth is significantly faster than both Difflog and ALPS on the program analysis benchmarks and most benchmarks with invented predicates. This is because subtle interactions between candidate rules make them uniformly harder than the remaining benchmarks in the suite, so that both ALPS and Difflog take significant amounts of time. These results demonstrate the effectiveness of provenance-guided constraints in rapidly reducing the size of the search space.

Finally, we note an interesting observation while running ProSynth on the 1-object-type benchmark: the choice of meta-rules was insufficient to encode the target program, so that the benchmark was actually unsynthesizable. Difflog was unable to recognize this contradiction and timed out on the benchmark, while the committee of candidate programs maintained by ALPS became empty after 257 seconds, thus correctly recognizing that the benchmark was unsynthesizable. In contrast, ProSynth converges to an unsatisfiable synthesis constraint in less than one second.

Note that all three algorithms in Table 2—ProSynth, Difflog, and ALPS—are searching over the same space of candidate programs. In addition to their synthesis algorithm, ALPS also introduced the process of augmentation we discussed in Section 3.3. The meta-rules produced as a result of this process can be further concretized into a set of candidate rules. This formed the input to both Difflog and ProSynth.

Furthermore, the small number of candidate rules for some benchmarks (such as path, sql04 and sql13) is a result of maintaining this parity across all three tools. We also ran ProSynth on versions of the benchmarks with a much larger set of candidate rules: we will discuss these results in Figure 8 and Section 5.4.

5.3 Variance in Running Time

We repeatedly ran both ProSynth and Difflog on each benchmark program, and collected running times and other statistics which we present in Figure 7. This figure demonstrates one of our important observations, i.e., that in addition to the improvements in performance, ProSynth also exhibits significantly smaller variance and much greater predictability in running times. Furthermore, in the vast majority of benchmarks (37 of 40), the maximum running time of ProSynth is faster than the median running time of Difflog, further substantiating our claims of improved performance. Note that while ALPS is mostly deterministic, showing only minimal variance in running times, its absolute performance is slower than both Difflog and ProSynth.

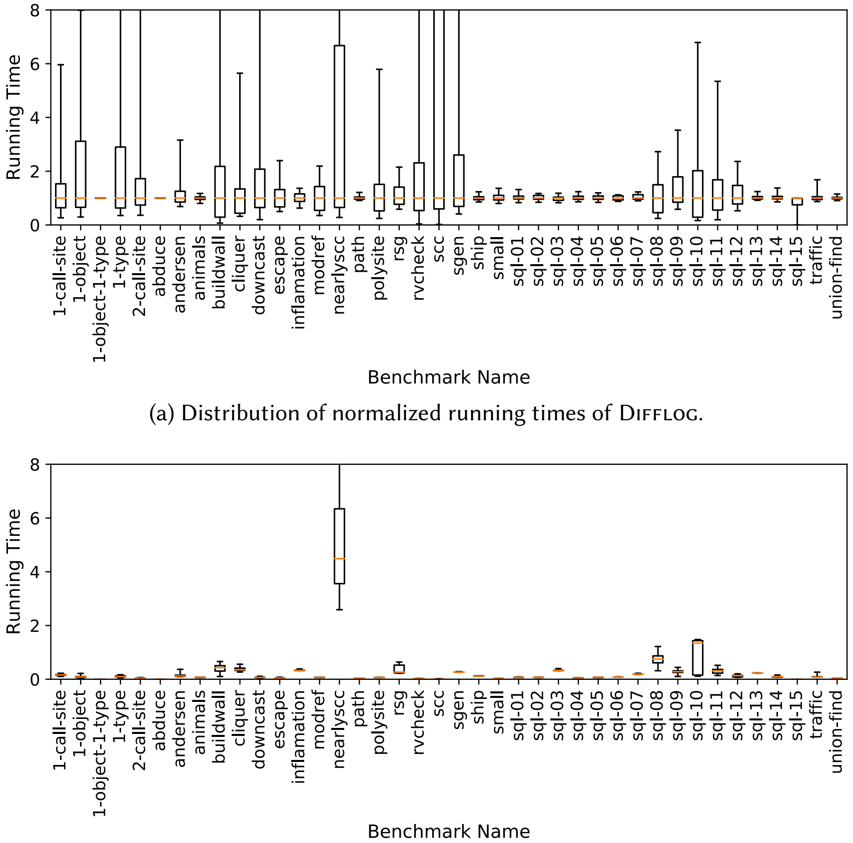
We point out the anomalous behavior of the sql10 and sql15 benchmarks, where ALPS and Difflog outperform ProSynth. A close analysis of these benchmarks shows that ProSynth and ALPS examine a similar number of programs. For sql10, ProSynth makes 635 calls to Z3 (in the

Table 2. Metrics summarizing the performance of ProSynth and its comparison to state-of-the-art approaches ALPS and Difflog. The first three columns indicate the number of candidate rules and the number of input and output tuples provided as part of the training data. For both ProSynth and Difflog, the statistics represent the median of 32 independent runs. All experiments were conducted with a timeout of 1 hour.

	# Candidate rules	# Tuples		Running time (seconds)			# Evaluated programs			
		Input	Output	ProSynth	ALPS	Difflog	ProSynth		ALPS	Difflog
							Z3	Soufflé		
abduce	38	12	8	<1	3	**timeout**	6	6	3,023	359,068
animals	76	50	64	<1	40	1	3	3	43,631	3
buildwall	178	30	4	31	67	71	133	1,255	44,779	74
cliquer	93	4	20	1	**timeout**	3	34	34	**timeout**	28
inflammation	67	640	49	<1	3	3	17	17	2,327	8
nearlyscc	166	5	17	25	1	6	318	1,281	558	17
path	5	7	31	<1	<1	<1	1	1	6	1
rsg	67	17	11	<1	**timeout**	2	9	9	**timeout**	5
samegen	166	7	21	2	12	6	21	21	984	9
scc	166	10	25	16	56	2,822	81	844	47,527	4,008
ship	64	15	5	<1	**timeout**	1	3	3	**timeout**	1
small	38	8	19	<1	**timeout**	1	1	1	**timeout**	4
traffic	33	12	2	<1	**timeout**	<1	6	6	**timeout**	4
unionfind	151	21	36	<1	**timeout**	2	1	1	**timeout**	1
1-call-site	96	28	4	3	104	20	19	165	105	36
1-object	41	40	13	1	350	14	8	70	350	27
1-object-type	12	48	6	<1	257	**timeout**	1	1	256	893,692
1-type	39	42	15	1	13	10	7	63	13	15
2-call-site	103	30	15	6	688	151	18	202	687	80
andersen	64	7	7	<1	27	2	6	6	53,005	5
downcast	359	89	175	23	1,622	342	51	500	181,463	75
escape	29	13	19	<1	6	2	8	8	4,802	71
modref	30	18	34	<1	2,816	4	4	4	1,375,527	34
polysite	325	97	27	7	84	114	52	52	29,463	68
rvcheck	67	74	2	27	195	1,228	342	3,313	72,952	78,415
sql01	26	21	2	<1	<1	1	6	6	17	3
sql02	12	3	1	<1	<1	<1	4	4	4	4
sql03	57	4	2	<1	<1	1	33	33	0	4
sql04	5	9	6	<1	<1	<1	3	3	13	3
sql05	9	12	5	<1	<1	<1	6	6	0	3
sql06	8	9	9	<1	<1	<1	3	3	32	3
sql07	39	5	5	<1	<1	1	15	15	0	1
sql08	91	6	2	4	1	5	82	296	0	52
sql09	40	6	1	<1	<1	2	14	60	7	15
sql10	688	10	2	248	44	184	636	2,781	504	189
sql11	58	30	2	7	1	22	236	501	936	1,540
sql12	22	36	7	<1	<1	2	12	40	109	25
sql13	7	17	7	<1	<1	<1	5	5	2	1
sql14	13	11	6	<1	56	<1	5	11	8	3
sql15	153	50	7	22	12	**timeout**	245	513	291	3,417

median case) while ALPS evaluates 778 candidate programs. Similarly, for `sql15`, ProSynth makes 245 calls to Z3 while ALPS evaluates 344 programs. ProSynth is implemented in Python and repeatedly invokes Soufflé as an external process, which reloads its EDB via filesystem calls in each iteration. In contrast, ALPS is implemented entirely in C++. This bottleneck will be eliminated by using the new Python interface to Soufflé which is currently under development.

Another curious outlier in these experiments is the case of `nearlyscc`. Informally, the benchmark was chosen from an introductory programming assignment, where the target relation $\texttt{nearlyscc}(x, y)$ is derivable iff either $\texttt{path}(x, y)$ is derivable or if $\texttt{path}(y, x)$ is derivable. Notice



(a) Distribution of normalized running times of Difflog.

(b) Distribution of normalized running times of ProSynth.

Fig. 7. Distribution of running times of Difflog and ProSynth across 32 runs for each benchmark. The running times were normalized around the median running time of Difflog for each benchmark. ProSynth exhibits much lower variability than Difflog in running times for a given benchmark. Also, in all but three benchmarks, the maximum running time of ProSynth is lower than the median running time of Difflog.

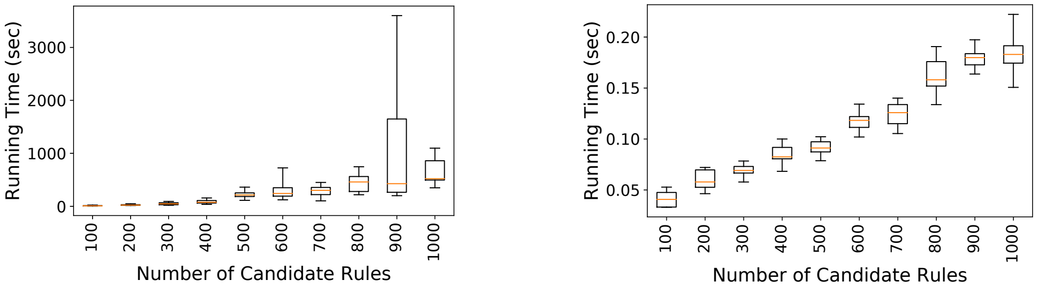
then that in, in addition to path, it requires the additional two rules:

$$\texttt{nearlyscc}(x, y) \text{ :- } \texttt{path}(x, y), \text{ and}$$
$$\texttt{nearlyscc}(y, x) \text{ :- } \texttt{path}(x, y).$$

This pair of rules interacts particularly well with the ALPS synthesis algorithm—where they form initial members of the committee, and where all other members are quickly evicted—so that ALPS significantly outperforms both Difflog and ProSynth on this specific benchmark.

5.4 Impact of Candidate Rules

To observe the impact of the candidate rule sets on synthesis times, we considered the exhaustive rule enumeration process of Section 3.3. We focused on the `scc` and `1-object-1-type` benchmarks and generated all candidate rules with at most 3 literals in their bodies, and considered subsets of them of varying sizes. We ran ProSynth multiple times on each subset and present the results in Figure 8. Notice that the variance rises somewhat quickly with the increasing number of candidate



(a) `scc`.

(b) `1-object-1-type`.

Fig. 8. Performance of ProSynth scaling with number of candidate rules on the `scc` and `1-object-1-type` benchmarks. Each set of candidate rules was randomly selected from a large set of generated candidate rules, and run 8 times with ProSynth.

rules, due to the larger search space. However, close inspection of the search logs reveals that the number of CEGIS iterations rises more slowly, indicating the effectiveness of why- and why-not provenance in extracting information and restricting the size of the search space. Finally, notice that the median running time of ProSynth rises only modestly with the number of candidate rules, demonstrating that adding extra candidate rules does not significantly impact the ability of ProSynth to find a solution.

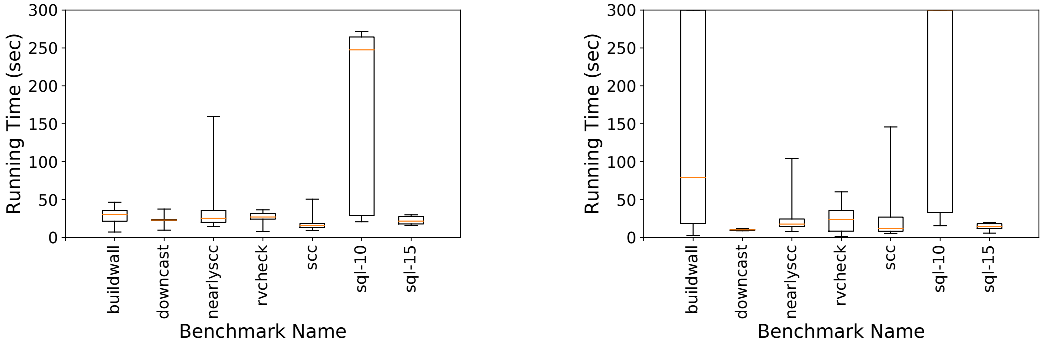
5.5 Impact of Optimizations

Our final experiment was to determine the impact of the optimizations on why-not provenance from Section 4.2. For each benchmark, we considered the number of CEGIS iterations made by ProSynth with $\overline{\text{prov}}_\Delta(t, P^-)$ and $\overline{\text{prov}}_S(t, P^-)$ respectively. We present the results for each of the large benchmark problems (those requiring more than 10 seconds to synthesize) in Figure 9, and for the remaining benchmarks in the Appendix. Observe that $\overline{\text{prov}}_\Delta(t, P^-)$ requires significantly fewer iterations to converge, and therefore learns better overall from each failed candidate program.

6 RELATED WORK

Our work on ProSynth follows a rich history of research in simplifying user interaction with complex data processing systems. In this section, we provide a brief survey of this work and categorize it into (*a*) research that aims to synthesize non-recursive table transformations such as SQL queries, (*b*) work on inductive logic programming (ILP), and (*c*) work on the synthesis of recursive logic programs. Furthermore, because of its central role in this paper, we also provide a brief overview of research on query provenance in databases, where we focus on provenance models and concrete implementations.

Synthesis of relational queries. Database researchers have long been interested in the challenge of making relational queries easier to compose by non-expert end-users. Examples for this research are the origins of SQL as the "*Structured English* Query Language" [Chamberlin and Boyce 1974], and the approximately contemporaneous development of Query-by-Example by [Zloof 1975]. The central challenge in systems which synthesize relational queries is to simultaneously determine both the hierarchical skeleton of the target query and the concrete predicates and constants inhabiting



(a) $\overline{\text{prov}}_{\Delta}(t, P^{-})$.

(b) $\overline{\text{prov}}_{S}(t, P^{-})$.

Fig. 9. Distribution of running times of ProSynth with (9a) and without (9b) delta-debugging inspired $\overline{\text{prov}}$ minimization. We show the distribution over 32 runs on benchmarks that take time greater than 10 seconds. See Table 3 in Appendix A for results for the remaining benchmarks.

operators such as select and join. One prominent recent approach is to view the candidate query skeletons as partial programs. For example, Morpheus [Wang et al. 2017] encodes partial programs as over-approximate constraints in an SMT solver. Another example is Scythe [Feng et al. 2017] that combines indistinguishability-based enumeration and prunes candidates using partial evaluation. Instead of analyzing examples of input-output data to synthesize the query, [Wang et al. 2018] applies symbolic provenance analysis to the query to synthesize input tables which satisfy some user-specified property, such as resulting in ill-formed output tuples or witnessing the inequivalence of two queries.

The problems solved by these papers are orthogonal to ours. First, the rules of a Datalog program interact in highly non-trivial ways. We noticed this in the example of Section 2, where both candidate programs $P_1 = \{r_3, r_6, r_7\}$ and $P_2 = \{r_1, r_2, r_3\}$ are valid solutions, but $P_1 \cup P_2 = \{r_1, r_2, r_3, r_6, r_7\}$ is not a solution. The central insight of our paper is to identify these composition-induced dependencies among rules and use them to guide the candidate search. Second, we assume that all constants appearing in the program are *uninterpreted* and that the only operation allowed between them is the implicit test for equality, $x = y$, across bound variables with the same name. However, this is insufficient in many practical situations, which require predicates (such as $x > 100$), aggregations (such as min and sum) and group-by operations. The efficient synthesis of these predicates is the main contribution of papers including Morpheus and Scythe. Hence, combining these approaches, i.e., provenance-guided skeleton synthesis with rich data types, including functions, predicates and aggregation, is an essential direction of future work.

Inductive Logic Programming. While ProSynth borrows some ideas from the field of inductive logic programming (ILP) [Raedt 2008], there are several key differences. First, ILP techniques usually learn relations, often probabilistic ones [Raedt and Kersting 2008], from large amounts of mined data, e.g. biological data [Muggleton 1999]. In contrast, ProSynth and other Datalog synthesis techniques infer a program from a *small representative set of examples*. Second, ILP techniques have traditionally not been well suited for the synthesis of recursive programs. In [Flener and Yilmaz 1999], Flener and Yilmaz survey techniques for recursive program synthesis using ILP. More recently, [Muggleton et al. 2015] present a recursive synthesis approach, based on executing meta-rules with an instrumented Prolog engine. However, a fundamental problem with synthesizing

Prolog programs is ensuring program termination. In their system, termination is achieved by relying on extensive user annotations in the form of lexicographic orders and other priority sequences on relations. Third, we employ a *complete* search strategy, while ILP techniques may fail to find a program even if one exists.

Recursive Datalog Synthesis. The problem of synthesizing recursive Datalog programs has been explored by some previous approaches. Zaatar [Albarghouthi et al. 2017] uses a constraint-solving algorithm, encoding SMT constraints that describe the output of candidate Datalog programs. However, Zaatar does not exploit the provenance information of output tuples, and ProSynth gains an advantage by utilizing provenance. Difflog [Si et al. 2019] employs continuous optimization techniques to synthesize Datalog programs. Using these techniques, Difflog can handle noise better than discrete approaches and is also able to synthesize approximate solutions for undecidable problems. ALPS [Si et al. 2018] is a syntax-guided approach for Datalog synthesis, using refinement techniques on the syntax of Datalog programs to generate a program. Zaatar and ALPS sit at opposite extremes of the spectrum of constraint-solving and enumerative-search techniques, respectively, which hinders their scalability. In contrast, ProSynth employs a hybrid of these two kinds of techniques in the CEGIS framework. Lastly, as our evaluation demonstrates, Difflog suffers from significantly higher variability in synthesis time compared to ProSynth.

Query Provenance. The concept of query provenance emerged from convergent attempts to debug database queries [Chiticariu and Tan 2006], to assess authority or uncertainty [Buneman et al. 2001; Green et al. 2007b], and to compute probabilities associated with individual tuples [Sarma et al. 2008]. A unified account of the concept was developed by [Green et al. 2007a], who observed that provenance has the mathematical structure of a semiring, and readily follows from replacing the Boolean operations of classical query evaluation with the operations of a different suitable data structure. We refer the reader to [Cheney et al. 2009] for a survey of the area.

Many previous approaches have explored the problem of synthesizing recursive Datalog programs. [Deutch et al. 2015, 2014; Köhler et al. 2012; Lee et al. 2019]. However, these techniques typically store the full provenance object during query evaluation. For example, [Köhler et al. 2012] stores the computation graph as an auxiliary relation during the evaluation, which may be many times larger than the output itself. [Deutch et al. 2015] and [Lee et al. 2019] reduce the impact of this storage by only storing information relevant to a particular query, determined even before query compilation. The weakness of these approaches is that the program needs to be re-evaluated for each new provenance query, and is therefore unsuitable in settings such as ours, where we rapidly seek the provenance of several different tuples. Therefore, the method recently implemented in Soufflé [Zhao et al. 2019], which provides minimal evaluation-time overhead, but requires a second pass for provenance reconstruction, is most suitable for our purposes.

In contrast to the sophisticated solutions available for why provenance, the theoretical challenges associated with why not provenance have limited the scope of practical implementations. [Lee et al. 2019] present an approach for computing the why not provenance for Datalog programs by enumerating all potential derivations of a tuple and showing the failure of each of them. While this approach is suitable for a human debugging a Datalog program, the large search space limits its practicality for use in automated systems such as ours. Thus, ProSynth borrows ideas from the area of delta-debugging [Zeller 1999] as a practical compromise to detect a small set of excluded rules which cause the non-production of a particular tuple.

7 CONCLUSION

We proposed a new approach to synthesize Datalog programs from input-output specifications. Our primary insight is to leverage query provenance to scale the CEGIS procedure in the setting wherein a SAT solver guesses the candidate Datalog program and a Datalog solver checks whether it meets the desired specification. We proposed novel algorithms to compute "*why*" and "*why not*" provenance information from a Datalog solver to efficiently learn the constraints for a SAT solver. We demonstrated the effectiveness of our approach in a tool called ProSynth on a variety of synthesis tasks. ProSynth is able to synthesize more programs than state-of-the-art approaches and runs an order of magnitude faster, often in under a second. Our reference implementation and experimental setup is publicly available at https://github.com/petablox/popl2020-artifact.

Our work points to several exciting future directions towards the synthesis of rule-based programs. First, our approach offers flexibility to support various extensions of Datalog, including as negation, aggregation and value construction. Each of these features requires additional syntactic constraints to be enforced on candidate programs, notably *stratification* in order to guarantee termination. Such constraints can be supported by replacing the SAT solver in our approach with an SMT solver. Another promising direction concerns the ability to handle noise in input-output specifications. This ability could be supported by relaxing the hard constraints generated in our approach and leveraging solvers for optimization extensions of SAT and SMT, such as MaxSAT and MaxSMT. We could also further extend the synthesis problem, for example, by requiring the smallest consistent program or the program with lowest computational complexity. Lastly, all existing approaches rely on template rules, which offer a syntactic scaffolding to guide synthesis. We plan to explore ways to relax the need for template rules upfront, for instance, by generating them on demand during the synthesis process.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Sasha Rubin for their insightful feedback. This research was supported in part by the U.S. Government through NSF awards #1737858 and #1836936 and ONR award #N00014-18-1-2021, by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project ARC DP180104030), and by a Facebook Research Award.

REFERENCES

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

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming (CP 2017)*. Springer, 689–706.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD 2013)*. IEEE, 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

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

Molham Aref, Balder ten Cate, Todd Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. ACM, 1371–1382.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the International Conference on Database Theory (ICDT 2001)*. Springer, 316–330.

Donald Chamberlin and Raymond Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control (SIGFIDET 1974)*. ACM, 249–264.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.

Laura Chiticariu and Wang-Chiew Tan. 2006. Debugging Schema Mappings with Routes. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*. VLDB Endowment, 79–90.

Andrew Cropper and Stephen Muggleton. 2015. Logical Minimisation of Meta-Rules Within Meta-Interpretive Learning. In *Inductive Logic Programming*. Springer, 62–75.

Jacek Czerniak and Hubert Zarzycki. 2003. Application of Rough Sets in the Presumptive Diagnosis of Urinary System Diseases. In *Artificial Intelligence and Security in Computing Systems*. Springer, 41–51.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Springer, 337–340.

Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2015. Selective Provenance for Datalog Programs Using Top-k Queries. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1394–1405. https://doi.org/10.14778/2824032.2824039

Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance. In *Proceedings of the 17th International Conference on Database Theory (ICDT 2014)*. OpenProceedings.org, 201–212. https://doi.org/10.5441/002/icdt.2014.22

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 422–436.

Pierre Flener and Serap Yilmaz. 1999. Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects. *The Journal of Logic Programming* 41, 2 (1999), 141–195. https://doi.org/10.1016/S0743-1066(99)00028-X

Todd Green, Grigoris Karvounarakis, Zachary Ives, and Val Tannen. 2007b. Update Exchange with Mappings and Provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*. VLDB Endowment, 675–686. http://dl.acm.org/citation.cfm?id=1325851.1325929

Todd Green, Grigoris Karvounarakis, and Val Tannen. 2007a. Provenance Semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2007)*. ACM, 31–40. https://doi.org/10.1145/1265530.1265535

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2016)*. Springer, 422–430.

Grigoris Karvounarakis, Zachary Ives, and Val Tannen. 2010. Querying Data Provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010)*. ACM, 951–962.

Emanuel Kitzelmann. 2010. Inductive Programming: A Survey of Program Synthesis Techniques. In *Approaches and Applications of Inductive Programming*. Springer, 50–73.

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative Datalog Debugging for Mere Mortals. In *Datalog in Academia and Industry*. Springer, 111–122.

Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2019. PUG: A Framework and Practical Implementation for Why and Why-not Provenance. *The VLDB Journal* 28, 1 (Feb. 2019), 47–71.

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

Stephen Muggleton. 1995. Inverse Entailment and Progol. *New Generation Computing* 13, 3 (Dec. 1995), 245–286. https://doi.org/10.1007/BF03037227

Stephen Muggleton. 1999. Scientific Knowledge Discovery Using Inductive Logic Programming. *Commun. ACM* 42, 11 (Nov. 1999), 42–46.

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

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2005. Abstract DPLL and Abstract DPLL Modulo Theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 36–50.

J. Ross Quinlan and Mike Cameron-Jones. 1995. Induction of Logic Programs: FOIL and Related Systems. *New Generation Computing* 13, 3 (Dec. 1995), 287–312. https://doi.org/10.1007/BF03037228

Luc De Raedt. 2008. *Logical and Relational Learning*. Springer.

Luc De Raedt and Kristian Kersting. 2008. *Probabilistic Inductive Logic Programming*. Springer, 1–27. https://doi.org/10.1007/978-3-540-78652-8_1

Anish Das Sarma, Martin Theobald, and Jennifer Widom. 2008. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE 2008)*. IEEE, 1023–1032.

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided Synthesis of Datalog Programs. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 515–527.

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs Using Numerical Relaxation. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. AAAI Press, 6117–6124.

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

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, 404–415. https://doi.org/10.1145/1168857.1168907

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 452–466.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2018. Speeding Up Symbolic Reasoning for Relational Queries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 157 (Oct. 2018), 25 pages.

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

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages Through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the Joint 7th European Software Engineering Conference and the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. Springer, 253–267.

Qiang Zeng, Jignesh Patel, and David Page. 2014. QuickFOIL: Scalable Inductive Logic Programming. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014), 197–208.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 239–248.

David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Provenance for Large-scale Datalog. arXiv:1907.05045 In submission.

Moshé Zloof. 1975. Query by Example. In *Proceedings of the National Computer Conference and Exposition (AFIPS 1975)*. ACM, 431–438.