From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems

AARON BEMBENEK, Harvard University, USA
MICHAEL GREENBERG, Stevens Institute of Technology, USA
STEPHEN CHONG, Harvard University, USA

Given a set of candidate Datalog rules, the Datalog synthesis-as-rule-selection problem chooses a subset of these rules that satisfies a specification (such as an input-output example). Building off prior work using counterexample-guided inductive synthesis, we present a progression of three solver-based approaches for solving Datalog synthesis-as-rule-selection problems. Two of our approaches offer some advantages over existing approaches, and can be used more generally to solve arbitrary SMT formulas containing Datalog predicates; the third—an encoding into standard, off-the-shelf answer set programming (ASP)—leads to significant speedups ($\sim 9\times$ geomean) over the state of the art while synthesizing higher quality programs.

Our progression of solutions explores the space of interactions between SAT/SMT and Datalog, identifying ASP as a promising tool for working with and reasoning about Datalog. Along the way, we identify Datalog programs as monotonic SMT theories, which enjoy particularly efficient interactions in SMT; our plugins for popular SMT solvers make it easy to load an arbitrary Datalog program into the SMT solver as a custom monotonic theory. Finally, we evaluate our approaches using multiple underlying solvers to provide a more thorough and nuanced comparison against the current state of the art.

Additional Key Words and Phrases: program synthesis, Datalog, inductive logic programming, answer set programming, satisfiability

ACM Reference Format:
Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2018. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems. In . ACM, New York, NY, USA, 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Datalog is a simple—but surprisingly useful—logic programming language. A program consists of a set of inference rules, and evaluation amounts to running these rules to a fixed point, making all possible inferences over both the initial data and any derived data. Despite its simplicity (or thanks to it), Datalog has found use in such domains as program analysis [Bravenboer and Smaragdakis 2009; Reps 1995; Scholz et al. 2016; Whaley and Lam 2004], networking [Loo et al. 2006; Ryzhyk and Budiu 2019], distributed systems [Alvaro et al. 2010a,b], and access control [Dougherty et al. 2006; Li and Mitchell 2003].

As interest in Datalog has grown, so has interest in Datalog program synthesis, where the task is to synthesize a Datalog program—i.e., a set of inference rules—that satisfies some specification, such as an input-output example. One popular strategy consists of two parts: first, generate a set of candidate Datalog rules; second, select a subset of these rules that meets the specification. A range of techniques have been proposed to solve the second phase of this approach (known as the

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

synthesis-as-rule-selection problem) including bidirectional search driven by query-by-committee [Si et al. 2018], numerical relaxation [Si et al. 2019], and counterexample-guided inductive synthesis (CEGIS) [Raghothaman et al. 2020]. These techniques have led to steady improvements on Datalog synthesis problems from various domains, including knowledge discovery, program analysis, and relational algebra.

The state-of-the-art solution for this problem is ProSynth [Raghothaman et al. 2020]. It uses CEGIS [Solar-Lezama et al. 2006], where a SAT solver proposes a selection of rules, and then a Datalog solver is used to test whether that selection matches the specification. Datalog provenance is used to add blocking constraints to the SAT solver to guide it to a satisfying selection.

In this paper, we present a progression of tools for solving the synthesis-as-rule-selection problem that builds off of ProSynth. Each tool explores a different balance between SAT solving and Horn clause evaluation (Datalog evaluation/finding blocking clauses); each balance affects how effectively the SAT solver is able to explore the exponentially large space of possible solutions. The ultimate tool in our progression has substantial speedups ($\sim 9\times$ geomean) over ProSynth, while being able to produce higher quality rule selections.

MonoSynth. This tool is based on the observation that every Datalog program can be the basis of a monotonic SMT theory. That is, it is possible to take an arbitrary Datalog program, and construct a (theoretically) efficient SMT theory from it. From a practical perspective, this tool is a version of ProSynth that is more tightly integrated into an SMT solver: like ProSynth, it uses a Datalog solver to test rule selections and Datalog provenance to generate conflicts; unlike ProSynth, it is able to give the SAT solver incremental feedback and prune bad partial rule selections. Furthermore, it provides a general way to solve SMT formulas involving Datalog predicates, and is not limited just to Datalog program synthesis.

LoopSynth. Inspired by answer set programming (ASP) algorithms, this tool takes a radically different approach to integrating SAT solving and Horn clause evaluation. In ProSynth and MonoSynth, the SAT solver never sees the candidate rules; in this approach, ground (variable-free) versions of the candidate rules are given to the SAT solver. This provides the SAT solver more information to guide its search for a satisfying assignment. The Datalog solver is used to confirm that the SAT model does in fact solve the problem; if not, the tool generates *loop formulas* that are added to the SAT solver. Like MonoSynth, LoopSynth can be used to solve SMT formulas involving Datalog predicates more generally, and it is not limited to just Datalog program synthesis.

ASPSynth. Going a step beyond LoopSynth, this tool encodes Datalog synthesis-as-rule-selection problems directly into ASP. Conceptually, the ASP solvers we use mix Horn clause evaluation and SAT solving, just like the other tools. However, by tightly integrating the two paradigms in a unified search procedure, ASP is able to more effectively and efficiently explore the solution space.

We perform a careful evaluation of all four tools on the ProSynth benchmark suite. To account for performance artifacts introduced by SAT solver internals, we evaluate versions of ProSynth, LoopSynth, and MonoSynth built on top of both Z3 [Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011], and ASPSynth built on top of Clingo [Gebser et al. 2011b] and WASP [Alviano et al. 2013]. We ultimately find that MonoSynth and LoopSynth can lead to some improvements over ProSynth, although the results are inconsistent and depend on the solver. However, ASPSynth-Clingo is a dominant approach. Its performance comes both from the fact that it is a single, tightly integrated system from an engineering perspective, and from the fact that this tight integration allows it to search the solution space while encountering fewer conflicts.

The key insight behind this success is a shift of perspective: a shift from seeing Datalog as a standalone (monotonic) logic programming language—a common (and reasonable) view in the programming language community, where Datalog has garnered much deserved attention in recent years—to seeing it as just a fragment of a more expressive (nonmonotonic) logic programming discipline, i.e., ASP. This perspective shift enables a move from SMT solvers—the constraint solving paradigm of choice in the programming language community—to ASP solvers, highly sophisticated tools that have received significant attention in the logic programming and artificial intelligence communities, but little in the programming language community.

1.1 Contributions

- We provide a progression of three new approaches for the Datalog synthesis-as-rule-selection problem, all of which explore different ways of combining SAT solving and Horn clause evaluation (Sections 4, 5, and 6). Two of our approaches provide a way to talk about Datalog predicates within SMT with applications beyond synthesis (Sections 4 and 5).
- We perform a thorough evaluation of these tools against ProSynth, the current state of the art, using multiple backend SAT solvers (Section 7). We demonstrate that the choice of solver can have a substantial impact on the relative performance of solver-based algorithms.
- We show the effectiveness of ASP for solving a problem of interest to the PL community: our ASP encoding of Datalog synthesis as rule selection achieves an order of magnitude ($\sim 9\times$ geometric mean) speedup over ProSynth, the state of the art (Section 7.4). Our work provides a hands-on overview of ASP techniques for the PL community, covering both loop-formula-based SAT encodings (Section 5) and direct solving approaches (Section 6).
- We explicate weaknesses in the current framing of the Datalog synthesis-as-rule-selection problem and show how it can be generalized to a form of bounded model checking (Section 8).

2 DATALOG SYNTHESIS AS RULE SELECTION

Section 2.1 gives background on Datalog, Section 2.2 on the Datalog synthesis-as-rule-selection problem. Section 2.3 overviews a baseline approach and our progression of solutions.

2.1 Datalog

A Datalog [Gallaire and Minker 1978; Green et al. 2013] program P is a set of inference rules over predicates p on vectors of terms $\mathbf{t}$, where each rule is a Horn clause:

$$p_0(\mathbf{t_0})\ \texttt{:-}\ p_1(\mathbf{t_1}),\ \ldots,\ p_n(\mathbf{t_n}).$$

A *term* t is either a constant c or a variable X (we use boldface $\mathbf{c}$ and $\mathbf{X}$ to refer to vectors of constants and variables, respectively). The atom $p_0(\mathbf{t_0})$ is the *head* of the rule; the remaining atoms make up the *body* of the rule. Variables in the head must occur in the body (which is allowed to be empty). Intuitively, each rule can be read as an implication from right-to-left: universally quantifying variables, the conjunction of the body atoms imply the head atom.

A Datalog program P's semantics is defined over a set of input facts, i.e., ground atoms (the extensional database, or EDB). If we have inputs facts I, then $P(I)$ is defined as the least fixed point of the rules in P given the initial facts in I, typically computed using semi-naive evaluation. The resulting set of facts is the output (the intensional database, or IDB).[1] Viewed as a function from EDBs to IDBs, P is positively monotonic: if $I \subseteq J$, then $P(I) \subseteq P(J)$. Alternatively, from a model-theoretic perspective, the meaning of a Datalog program coupled with an EDB is the least Herbrand model of the EDB and the rules of the program viewed as logical implications.

Datalog evaluation can be efficient, and is amenable to a wide variety of powerful optimizations (e.g., parallelism, goal-directed search). While not as expressive as general purpose logic

[1] We assume throughout that, for a given program, EDB predicates and IDB predicates are disjoint.

programming (e.g., Prolog), Datalog's balance of expressivity and performance make it a popular implementation choice in certain domains, such as static analysis [Bembenek et al. 2020; Bravenboer and Smaragdakis 2009; Flores-Montoya and Schulte 2020; Grech et al. 2019, 2018; Guarnieri and Livshits 2009; Jordan et al. 2016; Livshits and Lam 2005; Reps 1995; Scholz et al. 2016; Tsankov et al. 2018; Whaley and Lam 2004].

2.2 Synthesizing Datalog

Datalog program synthesis is the task of synthesizing a set of Datalog rules that satisfies some specification. We build upon a line of Datalog synthesis work that begins with ALPS [Si et al. 2018], which developed a methodology for generating a set of candidate Datalog rules from meta-rules. Given this set, ALPS chooses a subset that has the right behavior on an input-output example. This task—filtering a set of candidate Datalog rules for a subset that meets a specification—has become known as *synthesis as rule selection*. ALPS solves the synthesis-as-rule-selection problem using a bidirectional search strategy driven by query-by-committee. A line of follow-up work focuses exclusively on the synthesis-as-rule-selection problem (assuming the candidate rules generated by ALPS as given), using techniques inspired by numerical relaxation [Si et al. 2019] and, most recently, counter-example guided inductive synthesis [Raghothaman et al. 2020]. Our algorithms build on this last approach.

Synthesis as Rule Selection. Given (a) a sample input along with positive and negative output tuples and (b) a set of candidate rules, the task is to produce (c) a subset of the candidate rules that produces all the positive tuples and none of the negative ones. Formally, let I be the input tuples (the EDB), $\mathcal{T}^+_{exp}$ be the set of positive output tuples (a subset of the IDB), and $\mathcal{T}^-_{exp}$ be the set of negative output tuples. If P_{all} is the set of all candidate rules, we must select rules $P \subseteq P_{all}$ such that $\mathcal{T}^+_{exp} \subseteq P(I)$ and $\mathcal{T}^-_{exp} \cap P(I) = \emptyset$ (that is, all positive outputs are present, and no negative ones).

$$
\begin{aligned}
I &= \{\mathsf{edge}(1,2),\ \mathsf{edge}(2,1),\ \mathsf{edge}(2,3)\} \\
\mathcal{T}^+_{exp} &= \{\mathsf{path}(1,1),\ \mathsf{path}(1,2),\ \mathsf{path}(1,3), \\
&\qquad \mathsf{path}(2,1),\ \mathsf{path}(2,2),\ \mathsf{path}(2,3)\} \\
\mathcal{T}^-_{exp} &= \{\mathsf{path}(3,1),\ \mathsf{path}(3,2),\ \mathsf{path}(3,3)\} \\
\text{rule } 0 &= \texttt{path(X,Y) :- edge(Y,X).} \\
\text{rule } 1 &= \texttt{path(X,Y) :- edge(X,Y).} \\
\text{rule } 2 &= \texttt{path(X,Y) :- edge(X,Z), path(Z,Y).}
\end{aligned}
$$

Fig. 1. The synthesis-as-rule-selection problem involves choosing a subset of candidate rules that fits an input-output example (rules 1 and 2 in this case).

For example, the specification for synthesizing graph transitive closure might include an example graph I defined by edge predicates and sets $\mathcal{T}^+_{exp}$ and $\mathcal{T}^-_{exp}$ consisting of path predicates; the set P_{all} would consist of rules defining path in terms of edge and itself (Figure 1).

2.3 Four Approaches

We present four approaches to the Datalog synthesis-as-rule-selection problem: the state of the art along with three of our own. We diagram each approach using the following visual language: SAT solving is in blue and Datalog evaluation is in orange; dashed arrows indicate one-off interactions, while solid arrows indicate repeated interactions; components are labeled as standalone executables (.exe), libraries (.lib), or interpreted Python programs (.py).

We first present ProSynth [Raghothaman et al. 2020], the state of the art. ProSynth connects a SAT solver and a Datalog solver enriched with why and why-not provenance (Section 3; Figure 2), making many calls to Datalog and many calls to SAT in a counterexample-guided synthesis (CEGIS) loop [Solar-Lezama et al. 2006].

All three of our approaches reduce the overhead of this CEGIS loop; that is, they reduce the number of back-and-forth calls to resolve bad guesses made in the synthesis process.

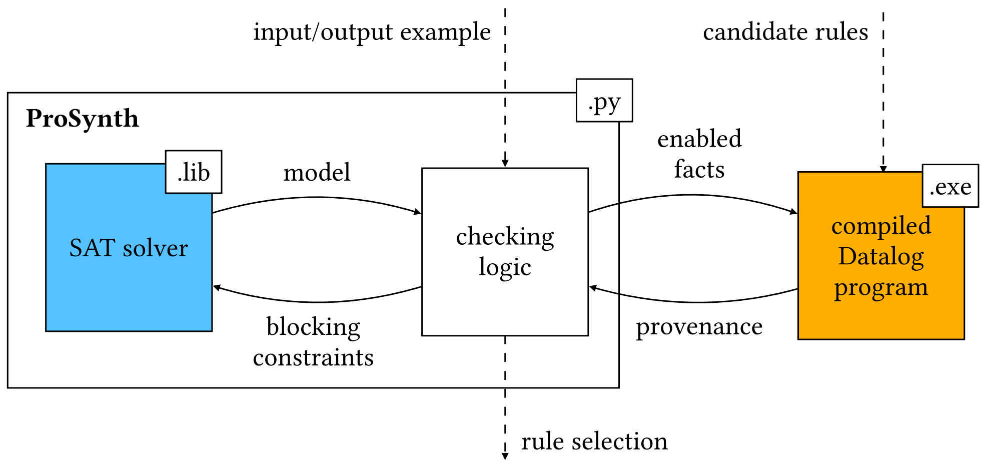


Fig. 2. ProSynth is the state-of-the-art tool for solving Datalog synthesis-as-rule-selection problems. Based on the CEGIS framework, it passes information back and forth between SAT solving and Datalog evaluation (Section 2.3 defines the visual language).

Our first approach, MonoSynth, takes advantage of Datalog's monotonicity to treat Datalog predicates as SMT predicates (Section 4; Figure 3). MonoSynth makes a single call to SMT, but many calls to Datalog evaluation; like ProSynth, it relies on provenance information.

Our remaining two approaches follow ideas from answer set programming (ASP). ASP solvers typically work in two phases: grounding and solving. Grounding eliminates all variables, yielding "ground" rules. Grounding is an expensive process, devolving in the worst case to explicit enumerations of all possible combinations of variable substitutions. In practice, grounders use various strategies to avoid plain enumeration. The solving process in ASP is a form of SAT search extended with mechanisms for ensuring that the resulting models satisfy logic programming semantics (i.e., are consistent with Horn clause resolution).

The first of these two approaches, LoopSynth, is inspired by the notion of *loop formulas* from ASSAT [Lin and Zhao 2004] (Section 5; Figure 5). We encode a synthesis problem into an ASP program and use the Gringo grounder [Gebser et al. 2007] to produce ground rules. We then encode the resulting system into SAT using the Clark completion [Clark 1977], asserting loop formulas to ensure that we generate a stable model (i.e., a valid solution). Where MonoSynth calls SMT just once, LoopSynth calls SAT incrementally. MonoSynth potentially calls Datalog multiple times per CEGIS iteration; LoopSynth grounds the program—which is tantamount to producing a superset of the IDB, i.e., doing Datalog evaluation—and then calls Datalog once per CEGIS iteration. LoopSynth does not use any Datalog provenance.

Finally, we encode the problem *directly* into ASP in the approach we call ASPSynth (Section 6; Figure 8). This approach is the simplest and the most efficient: there is no back-and-forth, as we just ship the encoded problem to an off-the-shelf ASP grounder and solver. Moreover, encoding into ASP makes it easy to specify a cost measure on rules, improving the quality of our solutions.

3 THE STATE OF THE ART: PROSYNTH

ProSynth is the state-of-the-art tool for solving the Datalog synthesis-as-rule-selection problem [Raghothaman et al. 2020]. Its technique is based on counterexample-guided inductive synthesis, and involves passing information back and forth between a SAT solver and a Datalog solver (Figure 2).

ProSynth takes as input a Datalog program consisting of the rules in P_{all}, except that each rule has been extended with a new premise rule(n) that identifies it as the nth rule. If the fact rule(n) is set in the EDB, then that candidate rule is enabled; if not, it can never fire. Furthermore, each rule in P_{all} is associated with a boolean variable in the SAT solver.

The SAT solver produces a guess of which rules to enable (i.e., the ones whose boolean variables are true in the produced model); the corresponding rule(n) facts are enabled in the Datalog program, which is run on the example input. By default, ProSynth uses Soufflé's compiled mode [Jordan et al. 2016] for Datalog evaluation: Datalog programs compile into C++, which must be compiled in turn. If the output of the Datalog program does not match the specification, ProSynth's algorithm uses *provenance* [Woodruff and Stonebraker 1997] to generate blocking constraints to pass back to the SAT solver. If an undesirable tuple appears in $P(I)$, then *why provenance* [Buneman et al. 2001] can indicate the rules which led to that tuple—turning off one or more of those rules hopefully excludes the tuple. (ProSynth uses Soufflé's provenance [Zhao et al. 2020], which doesn't generate multiple paths—so overdetermined tuples may not disappear after turning off the rules [Green et al. 2007].) If a desired tuple is missing from $P(I)$, then *why-not provenance* [Herschel et al. 2009; Lee et al. 2019] can indicate the rules which could have led to that tuple—turning on one of those rules hopefully generates the tuple. To generate why-not provenance, ProSynth uses a technique inspired by delta debugging to filter the set of all disabled rules for a smaller subset of rules; unless a rule in this subset is enabled, some particular desired tuple will not be derived. This process involves multiple invocations of Soufflé. ProSynth hands these constraints back to the SAT solver, which then produces a new guess; the process loops until it finds a solution or exhausts the search space.

ProSynth's implementation expects $\mathcal{T}^+_{exp}$ to be *exhaustive*, i.e., it specifies not merely a subset of the desired output, but the output itself. Accordingly, they set $\mathcal{T}^-_{exp}$ to the complement of $\mathcal{T}^+_{exp}$ automatically. It would not be too difficult to alter their tool to allow for custom $\mathcal{T}^-_{exp}$.

4 TIGHTENING THE LOOP: DATALOG AS A MONOTONIC SMT THEORY

Our first algorithm builds directly upon ProSynth's approach and directly addresses two potential limitations stemming from ProSynth's loose integration of the SAT solver with the Datalog solver. From an engineering perspective, ProSynth entails the overhead of communication between multiple OS processes: ProSynth is itself written as a Python script using Z3's bindings, but calls out to Datalog programs compiled by Soufflé into separate executables. From an algorithmic perspective, the Datalog computation does not provide the SAT solver with any incremental feedback when it is in the process of choosing a rule selection: it generates counterexamples only after the SAT solver has guessed a *complete* selection of candidate rules (i.e., the SAT solver has marked every rule as "in" or "out"), even if the SAT solver made bad decisions early on in the rule selection that could not possibly lead to a solution.

Our approach addresses these limitations by extending SMT solvers with a theory of Datalog.[2] Following ProSynth, the theory uses Datalog why and why-not provenance to derive precise theory conflicts. Unlike ProSynth, the SAT solving, Datalog solving, and conflict-generation logic all occur within a single OS process. Furthermore, because this theory of Datalog is a *monotonic* SMT theory (Sections 4.1 and 4.2), the Datalog solver is able to provide counterexamples to the SAT solver based on *partial* rule selections (i.e., the SAT solver is still undecided about some rules), enabling the SAT

[2]This makes more precise the observation that ProSynth is in some respects DPLL(T) with a theory of least fixed points [Raghothaman et al. 2020].

solver to eagerly prune solution-free subtrees of the search space. Our implementation (Section 4.3) works in Z3 and CVC4 and straightforwardly encodes the synthesis problem (Section 4.4).

4.1 Background: Monotonic Theories

In lazy SMT solving [Nieuwenhuis et al. 2006; Sebastiani 2007]—the basis of two of the most popular SMT solvers [Barrett et al. 2011; Moura and Bjørner 2008]—the core SAT solver assigns truth values to theory predicates; if that assignment is not satisfiable from the theory solver's perspective, it forces the SAT solver to backtrack and come up with a new assignment. For this integration to be efficient, the theory solver should provide two operations: theory propagation and conflict generation. *Theory propagation* takes a partial assignment to theory atoms and infers theory literals that must be true given that assignment. *Conflict generation* takes a partial assignment to theory atoms that is unsatisfiable (from the theory's perspective) and produces a clause of assigned atoms that conflict; the core SAT solver learns this clause's negation.

For example, say that we have a theory consisting of predicates of the form $x < y$, where the intended interpretation of $<$ is integer less-than. If the SAT solver has assigned true to the atoms $x < y$ and $y < z$, the theory solver can propagate the atom $x < z$. If the SAT solver had already assigned false to the atom $x < z$, then the theory solver could generate the conflict $x < y \land y < z \land \neg(x < z)$. The SAT solver could learn its negation, i.e., $\neg(x < y) \lor \neg(y < z) \lor x < z$, eventually causing search to find a model that violates our intended interpretation of $<$ as less-than.

SMT works best when theories can (1) perform propagation given a small partial assignment, and (2) return small clauses during conflict generation. Bayless et al. [2015] identify the class of *monotonic theories* as satisfying both criteria, given an efficient way to decide concrete instances of the theory problem. A monotonic theory is a theory where the only sort is boolean, and all predicates are monotonic in the sense that, for any such predicate p, it is the case that

$$p(\ldots, b_{i-1}, 0, b_{i+1}, \ldots) \implies p(\ldots, b_{i-1}, 1, b_{i+1}, \ldots).$$

That is, if a predicate holds when a given bit is turned off, it will continue to hold if that bit is turned on.[3] An example monotonic theory is finite graph reachability: a predicate $path_{a,b}(edge_1, \ldots, edge_n)$ is true iff there is a path from the node a to the node b, given that edge i is included in the graph iff $edge_i$ is assigned true. This is intuitively monotonic: when we add an edge to the graph, we do not invalidate any path that existed before.

If there is an efficient algorithm for deciding concrete instances of problems in the theory, a monotonic theory will effectively perform theory propagation and conflict generation. Our graph reachability example fits the bill: given a concrete graph, we can just run standard graph algorithms. How do monotonic theories perform theory propagation and conflict generation?

Theory propagation. Given a partial assignment M, let M_B be the partial assignment restricted to the exposed boolean variables of the theory (e.g., the $edge_i$ in the graph reachability example). Let M_B^+ be the positive extension of the assignment; i.e., it assigns 1 to any boolean variable unassigned in M. Given a theory predicate p, if $M_B^+ \implies \neg p$ (as determined by our decision procedure for concrete instances), then $M \implies \neg p$, and we can propagate the literal $\neg p$. Similarly, using the negative extension M_B^-, if $M_B^- \implies p$, then we can propagate p.

Conflict generation. Given a conflict arising from atoms by the over/under-approximation scheme described above, there is always some theory atom p in the conflict such that the assignment to the exposed boolean variables and the assignment to the atom p are together unsatisfiable. In this case,

[3] This is the definition of a *positive* monotonic predicate; monotonic theories also allow *negative* monotonic predicates, which are analogously defined. Furthermore, monotonic theories are allowed boolean-valued function symbols; since the SMT-LIB standard [Barrett et al. 2016] conflates such functions with predicates, we will just refer to "predicates."

if $M_B^- \implies p$, then the positive atoms in M_B imply p (and can justify a conflict involving p), and if $M_B^+ \implies \neg p$, then the negative atoms in M_B imply $\neg p$ (and can justify a conflict involving $\neg p$).

It is sometimes possible to learn better clauses than these defaults, when the algorithm used to decide a concrete instance of the problem provides a witness for its decision. For example, the edges along the path from vertex a to vertex b provide a justification for the atom $path_{a,b}(edge_1, \dots, edge_n)$; since their corresponding $edge_i$ booleans are necessarily a subset of the enabled SMT booleans, they constitute a more precise justification for that atom.

Monotonic theories are commonly used for generative purposes. For instance, the monotonic theory of finite graph reachability has been used to generate graphs (assignments to the $edge_i$) meeting constraints on which vertices can (or cannot) reach each other [Bayless et al. 2015], as well as for synthesizing packets that are able to reach certain nodes in a virtual cloud network [Backes et al. 2019]; a monotonic theory of computation tree logic has been used to synthesize systems described by this logic [Klenze et al. 2016]; and a monotonic theory of s-t maximum flow has been used as part of a solver ensemble for generating virtual data center allocations [Bayless et al. 2020].

4.2 Datalog as a Monotonic Theory

Every Datalog program is a monotonic theory. Given a pure (negation-free) Datalog program and a set of potential EDB facts, it is possible to construct a monotonic SMT theory consisting of the output tuples of the Datalog program parameterized by the potential input facts. Consider some Datalog program with n possible extensional facts, enumerated as x_1 through x_n. Say that $p(\mathbf{c})$ is a possible derived fact. We construct an SMT predicate symbol $p_{\mathbf{c}}$ that has the type $\mathbf{Bool}^n \to \mathbf{Bool}$. A predicate of the form $p_{\mathbf{c}}(b_1, \dots, b_n)$ will be true in the theory if and only if $p(\mathbf{c})$ is derived by the Datalog program under the set of extensional facts $\{x_i \mid b_i = 1\}$. Furthermore, the predicate meets the requirement for a monotonic theory:

$$p_{\mathbf{c}}(\dots, b_{i-1}, 0, b_{i+1}, \dots) \implies p_{\mathbf{c}}(\dots, b_{i-1}, 1, b_{i+1}, \dots).$$

Intuitively, the b_i allow us to turn on and off extensional facts, and so the monotonicity of the SMT predicate follows from the monotonicity of the Datalog program, where if we derive $p(\mathbf{c})$ under some set of extensional facts I, it will also be derived under any set that includes I. Such a Datalog-based theory fits naturally into the monotonic theory framework:

Theory propagation. Perform the standard over/under-approximation scheme by making two calls to the Datalog solver. The first call computes the program under the negative extension, i.e., computes $P(I^-)$ where $I^- = \{x_i \mid M_B^-[b_i] = 1\}$. If an atom is in $P(I^-)$, then it is implied by the positive variables in M_B^-. The second call computes the program under the positive extension, i.e., computes $P(I^+)$ where $I^+ = \{x_i \mid M_B^+[b_i] = 1\}$. If an atom is not in $P(I^+)$, then its negation is implied by the negative variables in M_B^+.

Conflict generation. Using provenance, we can do better than the default conflict justification scheme, which returns the positive predicates in M_B^- or the negative predicates in M_B^+.

(1) If a theory atom $p_{\mathbf{c}}(b_1, \dots, b_n)$ holds, the fact $p(\mathbf{c})$ must be present in $P(I^-)$; extract its provenance. Let D be the set of input facts appearing in a derivation of $p(\mathbf{c})$. As a justification for $p_{\mathbf{c}}(b_1, \dots, b_n)$, return the set $\{b_i \mid x_i \in D\}$, a subset of the positive variables in M_B^-.
(2) If a theory atom $\neg p_{\mathbf{c}}(b_1, \dots, b_n)$ holds, the fact $p(\mathbf{c})$ must not be present in $P(I^+)$. Use ProSynth's delta debugging technique to find a subset D of the facts in the set $I - I^+$, one of which must be enabled for $p(\mathbf{c})$ to be derived. Return the set $\{b_i \mid x_i \in D\}$ as a justification for $\neg p_{\mathbf{c}}(b_1, \dots, b_n)$; this will be a subset of the negative variables in M_B^+.

The ability to take an arbitrary Datalog program (equipped with a set of possible input facts) and turn it into a monotonic SMT theory has several potential uses outside Datalog synthesis. First, it provides a lightweight and declarative way to prototype monotonic theories that can be phrased as Datalog programs: just write the Datalog program, and then rely on our framework to perform theoretically efficient propagation and conflict generation. For example, the monotonic theory of finite graph reachability can be written as a three-rule Datalog program. While this implementation is certainly less efficient than a handwritten SMT theory, it requires much less investment than writing a full-fledged theory or custom propagator.

Second, our framework provides a way to solve SMT formulas that refer to Datalog predicates. Given that synthesis is the primary use of monotonic theories, we suspect that Datalog-based monotonic theories could be used in synthesis problems where part of the problem is phrased in Datalog and part is phrased in other SMT theories, as in the following example:

Example 4.1. Consider the task of finding whether there is a sequence of API calls that induces a system to reach a bad state, such as sensitive information being leaked on a public channel. This is in general a tough task, as there is an infinite number of possible sequences. We can test random sequences, but this is unlikely to find an interesting one; on the other hand, a more comprehensive testing strategy like symbolic execution is unlikely to scale, as there is an explosion of possible paths whenever the executor chooses which API call to make next. In contrast, our approach will combine the SMT theory of sequences (supported by Z3 and CVC4) and a Datalog-based monotonic theory to synthesize potentially interesting sequences of API calls that can then be checked using a more precise technique (like symbolic execution).

As the backend for our monotonic theory, we will use a Datalog-based analysis that computes an over-approximation of whether a sequence of API calls can lead to private information reaching a public sink (perhaps an extension of the taint analysis for Java of Livshits and Lam [2005]). This analysis is parameterized by the allowed orders of API calls; in particular, it takes as input facts of the form $\mathsf{start}(f)$ (indicating that f can be the first API call in the sequence) and $\mathsf{next}(f, g)$ (indicating that a call to f can be immediately followed by a call to g). As output, the analysis produces a fact error if the desired safety property might be violated given the allowed orders of API calls. Our monotonic SMT theory exposes these facts as SMT-level booleans.

By constructing SMT assertions that connect the Datalog facts to constraints from the SMT theory of sequences, we are able to synthesize interesting API call sequences: sequences that our Datalog analysis tells us might lead to an error condition, and which also meet any additional constraints we put on them using the theory of sequences (such as matching a regular expression). From the theory of sequences, we use the predicates $\mathsf{seq.prefixof}(pre, s)$ and $\mathsf{seq.contains}(s, sub)$. Let s be an SMT variable that will be instantiated with our sequence of calls. For each API method f, we assert that the fact $\mathsf{start}(f)$ holds iff $\mathsf{seq.prefixof}(\text{“}f\text{”}, \mathsf{s})$ holds. For each pair of API methods f and g, we assert that the fact $\mathsf{next}(f, g)$ holds iff $\mathsf{seq.contains}(\mathsf{s}, \text{“}fg\text{”})$ holds. Finally, we assert that the derived fact error holds. If this set of assertions is unsatisfiable, we know that there is no sequence of API calls that can lead to the safety property being violated. If it is satisfiable, we can extract a model of s and test that sequence of calls. Furthermore, we can use additional operators from the theory of sequences to guide our search. For example, to force the sequence to be of length k, we can assert $\mathsf{seq.len}(\mathsf{s}) = k$; to restrict it so that f cannot appear after g in the sequence, we can use regular expressions (supported by Z3) and assert $\neg\mathsf{seq.in.re}(\mathsf{s}, \text{“}.* \, g \, .* \, f .*\text{”})$.

We believe that these types of SMT assertions referring to Datalog predicates could in principle be encoded as programs written in Formulog [Bembenek et al. 2020], a variant of Datalog with mechanisms for representing and reasoning about SMT formulas. However, this approach is unlikely to work well in practice (in fact, our Formulog encoding of synthesis as rule selection fails to scale

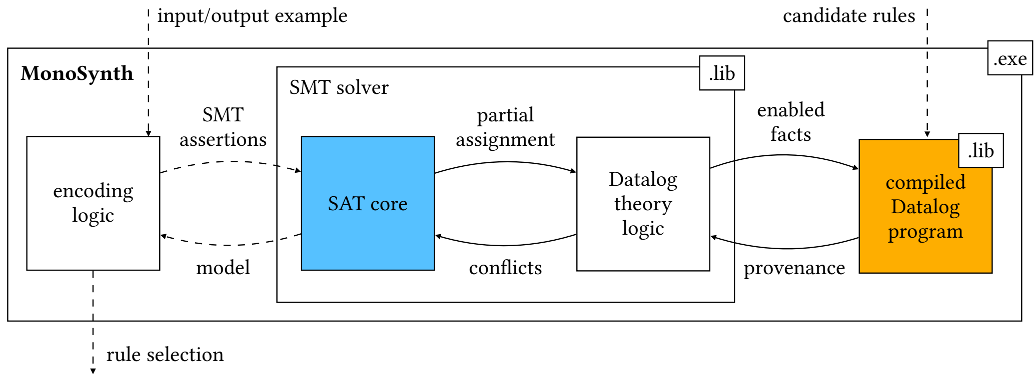


Fig. 3. MonoSynth uses a monotonic SMT theory of Datalog to solve the Datalog synthesis-as-rule-selection problem. The theory uses Datalog to generate conflicts on partial assignments proposed by the SAT core (Section 2.3 defines the visual language).

to any but the smallest ProSynth benchmarks). First, Formulog performs an exhaustive, saturation-based derivation process, instead of the guided search provided by SMT. Second, incremental SMT solving allows us to incrementally add new assertions (such as refinements to our desired solution) without restarting from scratch, whereas Formulog does not support incremental computation.

Like Formulog, constrained Horn clause (CHC) solving provides a way to solve a mix of Horn clauses and SMT formulas [Bjørner et al. 2015; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder and Bjørner 2012]. However, CHC solvers are typically more limited in the range of theories they support and—a more fundamental problem—they find classical models, while we need to interpret Horn clauses under a least model semantics to be consistent with Datalog.

4.3 Our Implementation

We have implemented two versions of this approach, one built as a custom theory in an extended version of CVC4, and one built as a user propagator on top of Z3. Both versions have essentially the same API, and very similar implementations. They expose an encoder class that is constructed with respect to a particular solver instance and a reference to a Soufflé program (which can be dynamically linked as a library).[4] The user passes a (ground) Soufflé tuple to the encoder; the encoder returns an SMT variable of boolean sort corresponding to that tuple. The user can assert arbitrary SMT formulas containing those boolean variables; the backend of the encoder makes sure that any models computed by the SAT core are acceptable under Datalog semantics, following the monotonic theory approach described in the previous section. The encoder uses Datalog why and why-not provenance to generate conflict justifications. Our implementation of the delta-debugging technique proposed by Raghothaman et al. [2020] follows the algorithm as implemented in ProSynth, which differs somewhat from the algorithm presented in the paper, making fewer Datalog calls but returning less precise why-not provenance.

Our implementation uses several heuristics. First, it uses a primitive truth maintenance system [Doyle 1979; McAllester 1990] to cache justifications for theory atoms propagated under the

[4]In a preliminary version, we tried using the incremental Datalog solver Differential Datalog [Ryzhyk and Budiu 2019] instead of Soufflé, in an attempt to take advantage of the fact that we make many very similar Datalog calls. On the problems we tried, the overhead of incremental computation, coupled with the lack of support for provenance, seemed to outweigh the potential benefit of Differential Datalog; however, using incremental Datalog evaluation is worth further exploration.

positive and negative extensions of each partial assignment. Second, given a partial assignment, there might be multiple conflicts to report. The ProSynth implementation reports up to 30 why conflicts and one why-not conflict per specified output relation. The Z3 propagator API permits only a single conflict per partial assignment; for the sake of consistency, we use the same strategy in both our Z3 and CVC4 versions. In general, we try to avoid Datalog evaluation and return small conflicts. Finally, an important heuristic is how eagerly we try to discover conflicts. It is not necessary to check for conflicts on every partial assignment; instead, our implementations buffer assignments to the boolean parameters until a threshold has been met (discussed in Section 7).

4.4 Encoding Synthesis

We have built a tool, MonoSynth, that uses the theory of Datalog to solve Datalog synthesis as rule selection problems (Figure 3). Given a synthesis problem, it pulls in a Soufflé program of candidate rules (as described in Section 3) programmatically as a shared library and loads the example input into it. Using the theory interface, it encodes and asserts the desired and undesired output tuples as SMT terms. It also encodes the $\mathsf{rule}(n)$ tuples, leaving them unconstrained. Finally, it invokes the SMT solver to check for satisfiability and construct a model; MonoSynth outputs the rule selection corresponding to the $\mathsf{rule}(n)$ tuples whose SMT equivalents are true in the model.

4.4.1 Proof of Correctness. Let $\mathcal{P} = (I, \mathcal{T}^+_{exp}, \mathcal{T}^-_{exp}, P_{all})$ be a rule selection problem. Let $\mathcal{R} : P_{all} \to \mathbb{N}$ be an injective function numbering the candidate rules. Let $P_{\mathcal{R},I}$ be P_{all} rewritten so that (a) each rule $r \in P_{all}$ includes a numbering premise $\mathsf{rule}(\mathcal{R}(r))$ and (b) each atom in I is included as a bodiless rule.

LEMMA 4.2. *If $R \subseteq \{n \mid \exists r \in P_{all}, \mathcal{R}(r) = n\}$ and $I_R = \{\mathsf{rule}(n) \mid n \in R\}$ such that $\mathcal{T}^+_{exp} \subseteq P_{\mathcal{R},I}(I_R) \wedge \mathcal{T}^-_{exp} \cap P_{\mathcal{R},I}(I_R) = \emptyset$, then, $P_R = \mathcal{R}^{-1}(R)$ solves $\mathcal{P}$.*

PROOF. First, $P_R \subseteq P_{all}$. Second, $P_R(I) = P_{\mathcal{R},I}(I_R) - I$, giving $\mathcal{T}^+_{exp} \subseteq P_R(I) \wedge \mathcal{T}^-_{exp} \cap P_R(I) = \emptyset$. □

THEOREM 4.3. *Let M be an SMT model of the MonoSynth encoding of the problem $\mathcal{P}$ and E be the mapping from Datalog facts to SMT booleans. Let $R = \{n \mid E(\mathsf{rule}(n)) \in M\}$; then $\mathcal{R}^{-1}(R)$ solves $\mathcal{P}$.*

PROOF. Our encoding uses the theory corresponding to the program $P_{\mathcal{R},I}$, where $E(\mathsf{rule}(i)) = b_i$ for each relevant $\mathsf{rule}(i)$ and $E(p(\mathbf{c})) = p_{\mathbf{c}}(\mathbf{b}_i)$ for each expected or unexpected tuple $p(\mathbf{c})$. Since M is a model of the SMT assertions MonoSynth makes, under the theory for program $P_{\mathcal{R},I}$, it must be the case that $\mathcal{T}^+_{exp} \subseteq P_{\mathcal{R},I}(I_R) \wedge \mathcal{T}^-_{exp} \cap P_{\mathcal{R},I}(I_R) = \emptyset$ for $I_R = \{\mathsf{rule}(n) \mid n \in R\}$; by Lemma 4.2. □

4.4.2 Comparison to Previous Approaches. MonoSynth differs from ProSynth in several key ways. First, all computation happens within a single OS process. ProSynth uses a Python process for its logic and for the SAT solver; each Datalog call gets its own process. Second, MonoSynth takes advantage of Datalog's monotonicity, proactively reporting conflicts on partial candidate rule selections. ProSynth only discovers conflicts once the SAT solver produces a full rule selection. Reporting conflicts on partial selections is a potential boon in itself, but it also makes it cheaper for MonoSynth to generate why-not conflicts using ProSynth's delta-debugging technique. ProSynth must filter the set of all rules excluded from a full rule selection, but MonoSynth need only filter from the rules negatively assigned in the current partial assignment. Relatedly, the ProSynth implementation reports multiple conflicts at a time; MonoSynth reports a single conflict per partial assignment, as noted in Section 4.3.

MonoSynth generally achieves good speedups over ProSynth (Section 7). MonoSynth-Z3 achieves about a ~ 2× speedup on average over ProSynth-Z3 (min/median/geometric mean (geomean)/ max: 0.03×/2.07×/1.83×/21.94×), and MonoSynth-CVC4 achieves close to an order-of-magnitude speedup on average over ProSynth-CVC4 (0.74×/9.08×/9.06×/103.30×).

Table 1. Under the stable model semantics, an ASP program has zero, one, or more solutions (answer sets).

Example ASP program	Answer sets	Intuition
`p :- p.`	$\emptyset$	p cannot be used to justify itself
`p :- not q.`	{p}	because q is not justified, p is
`p :- not q. q :- not p.`	{p}, {q}	either p or q is justified (but not simultaneously)
`p :- not p.`	none	any solution would be inconsistent

5 ABANDONING MONOTONICITY: BORROWING IDEAS FROM SAT-BACKED ASP

MonoSynth takes advantage of Datalog's monotonicity to solve SMT formulas containing Datalog predicates. In this section, we instead take advantage of Datalog's embedding in answer set programming (ASP), a *nonmonotonic* programming paradigm (Section 5.1). We adapt an existing algorithm for ASP to use an off-the-shelf SAT solver to solve SMT formulas containing Datalog predicates (Section 5.2); we apply our solver to synthesis problems (Section 5.3).

5.1 Background: SAT-Backed ASP Solving

Answer set programming (ASP) is a logic programming discipline that solves certain classes of search problems [Brewka et al. 2011; Gelfond and Lifschitz 1988]. ASP sits between Datalog and Prolog in expressivity: ASP programs always terminate (unlike Prolog) but solve NP-hard problems (unlike Datalog, which is PTIME [Papadimitriou 1985; Vardi 1982]).

Although modern ASP systems support a rich language of additional features, at its simplest, ASP is syntactically Datalog with the addition of negation-as-failure body literals in the form not $p(\mathbf{t})$. The addition of negation causes semantic difficulties (a least Herbrand model is no longer guaranteed), and multiple solutions have been devised, such as stratified negation [Apt et al. 1988; Przymusinski 1988; Van Gelder 1989], the well-founded semantics [Van Gelder et al. 1991], and the stable model semantics [Gelfond and Lifschitz 1988], which is the foundation of ASP.

Under the stable model semantics, an ASP problem has zero, one, or more solutions, which are sets of ground atoms known as answer sets. An answer set is a restricted model: every atom true in the model must be justified, and that justification must not assume the atom itself (Table 1).

To determine if a set of ground atoms S is an answer set of a program P:

(1) Generate the ground program P^G; this contains the *ground* version of all the rules in P, i.e., versions of the rules where all variables have been replaced following all possible substitutions mapping variables to constants. This program is guaranteed to be finite assuming a finite universe of constants.
(2) Compute P^G_S, the Gelfond-Lifschitz transformation of P^G with respect to S. This is a revision of the ground program with respect to two rules: (a) Discard any ground rule containing a body literal not $p(\mathbf{c})$ for some fact $p(\mathbf{c}) \in S$; and (b) remove all negated body literals from all rules that are not discarded—any such negated literal must be not $p(\mathbf{c})$ for some $p(\mathbf{c}) \notin S$.
(3) See if S matches the least model of P^G_S. Note that P^G_S is negation-free, so it is guaranteed to have a least model that can be computed as $P^G_S(\emptyset)$ (recall that P^G_S is a function from EDBs to IDBs; Section 2.1). The set S is an answer set if it matches this least model—i.e., a fact $p(\mathbf{c})$ is in S iff it is true in the least model of P^G_S.

Datalog fits within ASP: every Datalog program, partnered with an EDB, can be interpreted as an ASP program that has a single answer set (corresponding to the union of the EDB and IDB).

```
p(Y) :- r(X, Y), not q(Y).    q(Y) :- r(X, Y), not p(Y).    r(a, b).
```

(a) An example ASP program, P.

```
p(a) :- r(a, a), not q(a).
q(a) :- r(a, a), not p(a).
p(a) :- r(b, a), not q(a).
q(a) :- r(b, a), not p(a).
p(b) :- r(a, b), not q(b).
q(b) :- r(a, b), not p(b).
p(b) :- r(b, b), not q(b).
q(b) :- r(b, b), not p(b).
r(a, b).
```

(b) P grounded into P^G.

```
p(a) :- r(a, a).
q(a) :- r(a, a).
p(a) :- r(b, a).
q(a) :- r(b, a).
p(b) :- r(a, b).
p(b) :- r(b, b).
r(a, b).
```

(c) The Gelfond-Lifschitz transformation P^G_S of P^G.

Fig. 4. Solving an ASP program.

Example 5.1. Consider the ASP program P (Figure 4(a)). Because there are two constants (a and b) and two variables (X and Y) there are four possible substitutions. Our ground program P^G contains them all (Figure 4(b)). If we choose $S = \{\mathrm{p(b)},\ \mathrm{r(a,b)}\}$ for the Gelfond-Lifschitz transformation (Figure 4(c)), then the least model of P^G_S is the set $\{\mathrm{p(b)},\ \mathrm{r(a,b)}\}$. Since $S = P^G_S(\emptyset)$, the set S is an answer set of P. The only other answer set of P is the set $\{\mathrm{q(b)},\ \mathrm{r(a,b)}\}$.

ASP solving typically happens in two stages: first, the input ASP program is ground using a dedicated “grounder” (Section 5.1.1); second, a stable model for the ground program is found using SAT-inspired techniques. Here we adapt the ASSAT algorithm [Lin and Zhao 2004] for computing answer sets using an off-the-shelf SAT solver (Section 5.1.2); we discuss alternatives in Section 6.

5.1.1 Grounding. The first step of most ASP solving procedures is to ground the source program. We need not produce the *exact* ground program, but rather one that is equivalent under the stable model semantics. Modern grounders [Calimeri et al. 2017; Gebser et al. 2011a] use semi-naive evaluation to perform partial evaluation, producing smaller ground programs. For example, Gringo [Gebser et al. 2011a, 2007] produces this simpler ground program on Example 5.1:

```
p(b) :- not q(b). q(b) :- not p(b). r(a, b).
```

Despite optimizations, grounding can be the major bottleneck for solving some ASP problems. It is worth noting that grounding takes a program in first-order logic and produces a program in propositional logic, since each ground atom $p(\mathbf{c})$ can be treated as a propositional variable.

5.1.2 ASSAT. The ASSAT algorithm [Lin and Zhao 2004] computes answer sets for ground programs using off-the-shelf SAT solving. Because the programs are ground, in what follows we will treat all atoms p, q as being propositional. Given a rule r in the form p :- $q_1, \ldots, q_n$, let $head(r)$ be p and $body(r)$ be either the set $\{q_1, \ldots, q_n\}$ or the conjunction $\bigwedge_i q_i$, depending on context.

The first step of ASSAT is to construct the Clark completion [Clark 1977] of the ground program, encoding the atoms of the ground program in propositional logic: for each ground atom p in the ground program P^G, construct the equation $p \equiv \bigvee\{body(r) \mid r \in P^G,\ head(r) = p\}$. The Clark completion of P^G is the set of all such equations derived from P^G.

ASSAT asserts the Clark completion to a SAT solver. If the SAT solver finds a model, it remains to be seen that the model is a stable model. Not every model of the Clark completion is an answer set: positive cyclic dependencies in the completion allow the SAT solver to use a conclusion as an assumption in its own derivation. For example, the program `p :- p.` has the Clark completion $\{\mathsf{p} \equiv \mathsf{p}\}$. A possible SAT model is $\{\mathsf{p}\}$; however, this is not an answer set. Given a SAT model M, ASSAT computes the least model of P^G_M, the Gelfond-Lifschitz transformation of the ground input program with respect to M. If this matches M, then M is an answer set. If not, ASSAT asserts new formulas to rule out this model (such as $\neg p$, in our example). ASSAT re-invokes the SAT solver, iterating until (a) the SAT solver's model is an answer set or (b) SAT fails to find a model at all.

What sort of assertions does ASSAT make to rule out non-stable models? Given a SAT model M, ASSAT finds the atoms in M that are not in the least model of the Gelfond-Lifschitz transformation P^G_M. It constructs a positive dependence graph for these atoms, where vertices are labeled with the ground atoms and there is an edge from atom p to atom q if there is a rule in P^G with p in the head and q in the body. For each strongly connected component in this graph, ASSAT asserts a *loop formula*. To create a loop formula for a loop L, ASSAT first generates the set of rules $R^-(L)$:

$$R^-(L) = \{r \mid r \in P^G,\ head(r) \in L,\ \neg(\exists q.q \in body(r) \land q \in L)\}$$

The set of rules $R^-(L)$ are those rules that (a) can derive the atoms in the loop and (b) have no body atoms in the loop. That is: the rules in $R^-(L)$ are an external justification for the atoms in the loop. The loop formula for a loop L, written $LF(L)$, is then:

$$LF(L) = \left[\neg \bigvee \{body(r) \mid r \in R^-(L)\}\right] \Longrightarrow \bigwedge_{p \in L} \neg p$$

Asserting $LF(L)$ forces the SAT solver to "turn off" the atoms in the loop, unless it "turns on" the body of a rule in $R^-(L)$.

Example 5.2. Say we have a ground program P^G with the rules {`p :- p.`, `p :- q.`}. The Clark completion is the set $\{\mathsf{p} \equiv \mathsf{p} \lor \mathsf{q}, \mathsf{q} \equiv \mathit{False}\}$ (the last equivalence is included because no rule has q at the head). Say that the SAT solver returns the model $M = \{\mathsf{p}\}$. This is not an answer set of P^G, since $\mathsf{p} \in M$ but $\mathsf{p} \notin P^G_M(\emptyset)$. The violating loop L consists just of p, and so $R^-(L) = \{$`p :- q.`$\}$. Thus, the loop formula $LF(L)$ is $\neg q \Longrightarrow \neg p$.

Some programs, known as tight programs [Fages 1994], have no cycles in their positive dependence graphs. For these programs, the SAT model of the Clark completion corresponds to an answer set. Non-tight programs may need exponentially many loop formulas (often fewer in practice).

5.2 Our Implementation

ASSAT is an algorithm for solving ASP problems. Our tool adapts it to solve SAT/SMT formulas containing Datalog predicates (and, more generally, ASP predicates). We implemented two versions in Python, one using Z3's bindings, one using CVC4's. Both versions expose an interface for an "encoder" that wraps around an SMT solver instance, and is constructed with respect to a Datalog program and a set of possible input tuples. The encoder uses the grounder Gringo [Gebser et al. 2011a, 2007] to ground the program; it encodes rules directly, while encoding each possible input tuple $p(\mathbf{c})$ as an ASP choice rule $\{p(\mathbf{c})\}$. The choice rule is shorthand for the rules $p(\mathbf{c})$ `:-` not q. and q `:-` not $p(\mathbf{c})$., where q—an atom not occurring elsewhere in the program—represents the choice to omit $p(\mathbf{c})$ from the answer set.

From the ground program generated by Gringo, our tool constructs the Clark completion and asserts it to the SAT solver. Clients can then pass a Datalog fact to the encoder and get out a boolean SAT variable corresponding to that fact, which can be used in arbitrary SAT formulas. The encoder exposes a method for checking satisfiability of the underlying solver state (modulo stable model semantics); it does this following the ASSAT algorithm of iteratively generating a model with the SAT solver, checking it with a Datalog solver, and then adding loop formulas if necessary.

Out of convenience, our implementation uses Gringo as a Datalog solver here; it could just as well use Soufflé. It reports just a single loop formula: the one for the smallest SCC in the positive dependence graph of ground atoms in the SAT model but excluded from the least model.

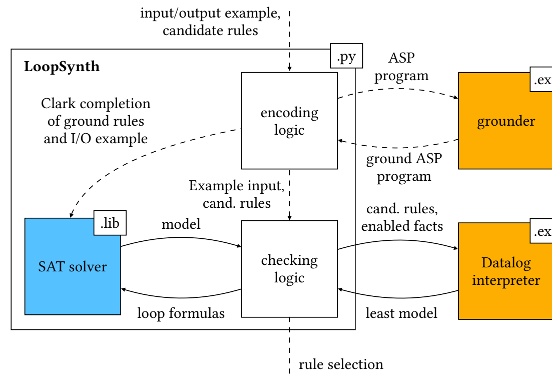


Fig. 5. LoopSynth uses the ASSAT algorithm to solve Datalog synthesis-as-rule-selection problems. The candidate rules are passed to a grounder, a SAT solver finds models of the Clark completion of the ground rules, and a Datalog interpreter is used to find loop formulas to refine the SAT solver's search (Section 2.3 defines the visual language).

5.3 Encoding Synthesis

Our tool LoopSynth uses the system described above to solve Datalog synthesis-as-rule-selection problems (Figure 5). Given a benchmark problem, it parses the Souffle program (containing the candidate rules) into an AST representation, augments it with the example inputs, and then passes it to the ASSAT-based encoder, along with the set of rule tuples. It then uses the encoder to encode each desired and undesired tuple from the problem specification, asserting each returned boolean SAT variable (or its negation) appropriately. The encoder is then queried for a model; a rule is considered to be selected if its corresponding SAT variable is true in the model.

5.3.1 Proof of Correctness. Let $\mathcal{P}$, $\mathcal{R}$, and $P_{\mathcal{R},I}$ be as defined in Section 4.4.1.

THEOREM 5.3. *Let M be a SAT model of the LoopSynth encoding of the problem $\mathcal{P}$ and E be the mapping from Datalog facts to SAT booleans. Let $R = \{n \mid E(\mathrm{rule}(n)) \in M\}$. Then $\mathcal{R}^{-1}(R)$ solves $\mathcal{P}$.*

PROOF. ASSAT [Lin and Zhao 2004] guarantees that M is an answer set S of the ASP program consisting of the rules in $P_{\mathcal{R},I}$ and the choice rules `{rule(n)}.` for $n \in \{n \mid \exists r \in P_{all}, \mathcal{R}(r) = n\}$. From this, we have $\mathrm{rule}(n) \in S \iff E(\mathrm{rule}(n)) \in M \iff n \in R$. Let $I_R = \{\mathrm{rule}(n) \mid n \in R\}$. It must be that $P_{\mathcal{R},I}(I_R) = \{p(\mathbf{c}) \mid p(\mathbf{c}) \in S \text{ and } p \text{ is an IDB predicate of } P_{\mathcal{R},I}\}$. LoopSynth's SAT assertions guarantee that $\mathcal{T}^+_{exp} \subseteq S$ and $\mathcal{T}^-_{exp} \cap S = \emptyset$. Thus, $\mathcal{T}^+_{exp} \subseteq P_{\mathcal{R},I}(I_R) \wedge \mathcal{T}^-_{exp} \cap P_{\mathcal{R},I}(I_R) = \emptyset$; the rest follows by Lemma 4.2. □

5.3.2 Comparison to Previous Approaches. LoopSynth's approach differs from the previous approaches in a few ways. First, the SAT solver actually sees the candidate rules, as they are encoded as part of the Clark completion asserted to the solver. Second, it calls Datalog less frequently: only once per SAT call. In ProSynth or MonoSynth, a single SAT assignment might result in multiple Datalog calls to generate provenance for conflict construction. Unlike MonoSynth, LoopSynth is not incremental, in that the SAT solver guesses a full rule selection before getting feedback.

LoopSynth has mixed performance results on the benchmark suite (Section 7). The CVC4 version achieves solid speedups over ProSynth-CVC4 (min/median/geomean/max: 0.00×/6.61×/3.14×/83.87×) but lags behind MonoSynth-CVC4 (0.00×/0.34×/0.35×/13.81×); it has the fastest average time out of all the CVC4-backed tools on 11/40 benchmarks. However, LoopSynth-Z3 struggles compared to both ProSynth-Z3 (0.00×/0.19×/0.24×/25.67×) and MonoSynth-Z3 (0.00×/0.10×/0.13×/7.71×). Even so, there are benchmarks for which it is faster on average than either tool.

6 NAMING THAT TUNE IN JUST ONE NOTE: A DIRECT ENCODING IN ASP

Existing approaches have always made more than one call to at least one system: ProSynth makes many calls to Soufflé and SMT (Section 3); the monotonic theory approach makes just one call to SMT, but many calls to Soufflé (Section 4); borrowing ideas from SAT-backed ASP yields an approach that makes far fewer calls to Datalog but still multiple calls to SAT (Section 5). If we can encode our problem directly in ASP, we can solve it all in just one go: a single call to grounding (cf. Datalog evaluation) and a single call to an ASP solver.

6.1 Encoding Synthesis

We phrase Datalog synthesis as an ASP program. Our encoding of rule selection and minimization is a specialization of the encoding that the ASP program synthesis tool ASPAL [Corapi et al. 2011] uses to instantiate skeleton ASP rules, modified to use auxiliary relations to encode output examples.

First, we list the specification's EDB facts as ASP facts. Next, for every relation p appearing in $\mathcal{T}^+_{exp} \cup \mathcal{T}^-_{exp}$, we create fresh predicate symbols p^+ and p^-. For every positive example $p(\mathbf{c}) \in \mathcal{T}^+_{exp}$, we state the fact $p^+(\mathbf{c})$. For every negative example $p(\mathbf{c}) \in \mathcal{T}^-_{exp}$, we state the fact $p^-(\mathbf{c})$.

We then add hard constraints to restrict the output of the synthesized program (Figure 6(a,b)). A *hard* constraint is a headless rule; any answer set that makes the body true is rejected. The constraint (C^+) requires that every positive example is derived, while (C^-) ensures that no negative example is derived. Together, these hard constraints force the selected rules to match the problem specification. Like LoopSynth, we can directly encode rule selection using ASP choice rules (Figure 6(c)): $\mathsf{rule}(n)$ may or may not be present in the answer set. In effect, the ASP solver selects the rules itself. So, the computed answer set answers our Datalog synthesis problem: rule n should be included in the synthesized Datalog program iff $\mathsf{rule}(n)$ is in the answer set.

$$\texttt{:-}\ p^+(\mathbf{X}),\ \text{not}\, p(\mathbf{X}).$$

(a) (C^+) includes positive examples.

$$\texttt{:-}\ p^-(\mathbf{X}),\ p(\mathbf{X}).$$

(b) (C^-) excludes negative examples.

$$\{\mathsf{rule}(n)\}.$$

(c) Choice rules select candidate rules.

Fig. 6. Encoding Datalog synthesis in ASP.

```
edge(1,2). edge(2,1). edge(2,3).
path⁺(1,1). path⁺(1,2). path⁺(1,3).
path⁺(2,1). path⁺(2,2). path⁺(2,3).
path⁻(3,1). path⁻(3,2). path⁻(3,3).
:- path⁺(X,Y), not path(X,Y).
:- path⁻(X,Y), path(X,Y).
path(X,Y) :- edge(Y,X), rule(0).
path(X,Y) :- edge(X,Y), rule(1).
path(X,Y) :- edge(X,Z), path(Z,Y),
             rule(2).
{rule(0)}. {rule(1)}. {rule(2)}.
```

Fig. 7. The ASP encoding for synthesizing graph transitive closure (Figure 1).

Example 6.1. We encode our earlier Datalog synthesis-as-rule-selection problem for synthesizing graph transitive closure (Figure 1) into ASP (Figure 7). The answer set for this program—which contains rule(1) and rule(2), but not rule(0)—answers the synthesis problem.

6.1.1 Handling Implicit Negative Examples. ProSynth's benchmark suite for Datalog synthesis does not provide negative examples; rather, $\mathcal{T}^+_{exp}$ is assumed to be exhaustive (i.e., derive only and exactly tuples in $\mathcal{T}^+_{exp}$). We can adjust our encoding to be exhaustive by replacing (C^-) with constraint ($\mathrm{C}^-_\forall$): $\texttt{:-}\ p(\mathbf{X}),\ \text{not}\, p^+(\mathbf{X})$. We use this encoding in the evaluation in Section 7.

6.1.2 Minimizing Rule Selection. There may be many selections of rules that generate $\mathcal{T}^+_{exp}$ and not $\mathcal{T}^-_{exp}$, and one might prefer a selection that minimizes the number or complexity of rules. For each rule n, we can add a fact $\mathsf{rule\_cost}(n, k)$, where k is a complexity measure of the rule. We then add a *weak* constraint (C^{min}):

$$:\sim \mathrm{rule}(X),\ \mathrm{rule\_cost}(X, C).\ [C, X]$$

The (C^{min}) constraint sets the cost to include a given rule in the answer set, as specified in its rule_cost fact (here, X is a grouping variable, entailing a cost of C for each assignment of X). To minimize the total number of premises across selected rules (a measure of solution complexity used by other logic program synthesis tools [Law et al. 2015]), we can set the k in rule_cost(n, k) to be the number of premises in rule n. None of the approaches presented previously in this paper attempt to minimize rule selection. In principle, it might be possible to do so by backing them with a MaxSAT/MaxSMT solver and weighting the variables corresponding to rule selection appropriately.

6.1.3 Proof of Correctness. Let $\mathcal{P}$, $\mathcal{R}$, and $P_{\mathcal{R},I}$ be as defined in Section 4.4.1.

THEOREM 6.2. *If S is an answer set of the ASPSynth encoding of the problem $\mathcal{P}$ and $R = \{n \mid \mathrm{rule}(n) \in S\}$, then $\mathcal{R}^{-1}(R)$ solves $\mathcal{P}$.*

PROOF. Let $I_R = \{\mathrm{rule}(n) \mid n \in R\}$. It must be that $P_{\mathcal{R},I}(I_R) = \{p(\mathbf{c}) \mid p(\mathbf{c}) \in S \text{ and } p \text{ is an IDB predicate of } P_{\mathcal{R},I}\}$. Constraints ($C^+$) and ($C^-$) guarantee that $\mathcal{T}^+_{exp} \subseteq S$ and $\mathcal{T}^-_{exp} \cap S = \emptyset$. And so, $\mathcal{T}^+_{exp} \subseteq P_{\mathcal{R},I}(I_R)$ and $\mathcal{T}^-_{exp} \cap P_{\mathcal{R},I}(I_R) = \emptyset$. The rest follows from Lemma 4.2. □

6.1.4 Comparison to Previous Approaches. ASPSynth (Figure 8) makes only one "Datalog" call (grounding) and one "SAT" call (ASP solving), unlike previous systems. It is less general than MonoSynth and LoopSynth, which can solve arbitrary SMT formulas containing Datalog predicates. However, modern ASP solvers make it easy (a) to avoid explicitly enumerating all undesired tuples (necessary in MonoSynth and LoopSynth) and (b) to encode solution minimization. Compared to the other tools, it encodes the entire synthesis problem—both the candidate rules and the specification—in a single form that can be solved all at once, allowing an incremental interaction between selecting candidate rules and checking the consequences of that selection against the specification.

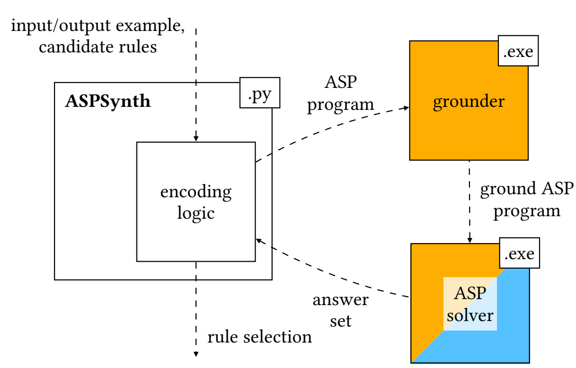


Fig. 8. ASPSynth encodes synthesis-as-rule-selection problems as ASP programs that flow to a grounder and then an ASP solver (dual shading indicates the combination of SAT search with logic programming semantics; Section 2.3 defines the visual language).

6.2 Implementation

We implement the direct encoding approach in two systems, ASPSynth-Clingo and ASPSynth-WASP, that respectively use the ASP solvers Clingo v5.4.0 [Gebser et al. 2011b] and WASP v2.0 [Alviano et al. 2013]. Both use the grounder Gringo v5.4.0 [Gebser et al. 2011a, 2007]. Clingo runs Gringo internally: technically, "Clingo" refers to the composite system consisting of Gringo and the solver Clasp [Gebser et al. 2012]. For ASPSynth-WASP, we run Gringo separately and pipe the results into WASP. Both Clingo and WASP are "native" ASP solvers: custom solvers that use SAT-like techniques, but do not discharge to an external SAT solver.

Direct ASP encodings have the best overall performance on the benchmark suite. WASP has ~3× average speedups over ProSynth-Z3 (min/median/geomean/max: 0.03×/2.63×/3.62×/186.10×)

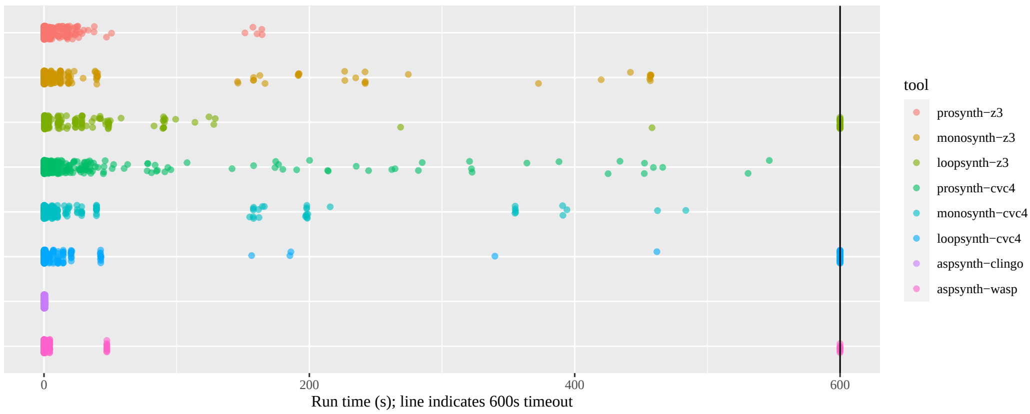


Fig. 9. Each approach has its moments, but the direct-to-ASP encodings have the best overall performance on the benchmark suite with few long-running outliers (ASPSynth-Clingo has none). Each dot in the chart represents a trial (400 trials/tool); dots are semi-transparent and height-jittered to give a sense of density.

and ~2× speedups over MonoSynth-Z3 (0.07×/1.64×/1.98×/247.98×). Clingo does even better: ~9× average speedups over ProSynth-Z3 (1.16×/7.50×/9.47×/372.19×) and ~5× speedups over MonoSynth-Z3 (0.33×/3.88×/5.17×/513.80×). The next section gives evaluation details.

7 EVALUATION

We empirically evaluate the approaches we have developed on the ProSynth benchmark suite. Our benchmark results report the mean of ten runs; we used an otherwise idle Ubuntu 20.04 AWS server with 32 vCPUs (clocked at 3.1 GHz), 128 GiB RAM, and 600 GiB of SSD. We use Z3 v4.8.15 and CVC4 v1.8. Each individual benchmark trial times out after ten minutes (600s). The benchmarks and experiment scripts are available in the paper artifact.

We use ProSynth as a baseline. We have slightly modified the version of ProSynth from the original paper artifact to record and print out additional statistics, and to randomize the names used for SMT symbols. The choice of SMT symbol names can have a substantial (and arbitrary) impact on solver performance, and we did not want ProSynth to be unfairly benefited or hurt by what is ultimately an arbitrary solver artifact. (We similarly randomize the SMT symbol names in our SMT-based tools.) Finally, we created an alternative version of ProSynth that uses CVC4 instead of Z3; this involved minor syntactic changes to move from Z3's Python bindings to CVC4's. Our version of ProSynth-Z3 performs on par with the original and significantly better than the CVC4 version (min/median/geometric mean (geomean)/max speedup: 0.68×/3.97×/5.27×/174.02×).

Each approach has cases it performs well on, but the direct ASP encodings—especially ASPSynth-Clingo—have the best overall performance on the benchmark suite (Figure 9; Table 7). The ASP encodings show overall fast performance. ASPSynth-Clingo is the best: it has *no* slow outliers and yields results on every single benchmark in under a second; when ASPSynth-Clingo isn't the fastest, it is a mere 0.02s slower. In what follows, we describe the benchmark suite (Section 7.1), break down results for each of our approaches (Section 7.2, 7.3, and 7.4), and evaluate ASPSynth-Clingo in a wider context (Section 7.5).

Table 2. ASPSynth-Clingo is the fastest tool on 26/40 benchmarks (the fastest time for each benchmark is in **bold**); when it is bested, the winner is one or both of the MonoSynth variants. Neither ProSynth nor LoopSynth is ever the fastest tool. If a tool timed out on all trials, we report its time as "TO"; if it timed out on some but not all trials, we precede its time with a "*" and count its timed-out trials as taking 600 seconds.

	*x*Synth-Z3 (s)			*x*Synth-CVC4 (s)			ASPSynth-*x* (s)	
Benchmark	Pro	Mono	Loop	Pro	Mono	Loop	Clingo	WASP
1-call-site	11.94	5.16	86.65	47.36	6.10	*560.18	**0.07**	4.25
1-object	0.81	0.61	1.16	4.76	0.34	0.41	**0.04**	0.07
1-object-1-type	0.05	**0.01**	0.39	0.13	0.05	0.15	0.03	0.04
1-type	0.98	0.64	3.21	4.28	0.46	1.18	**0.05**	0.12
2-call-site	5.23	18.72	23.68	35.61	17.22	10.59	**0.19**	0.77
abduce	0.06	**0.03**	0.39	2.02	0.04	0.16	**0.03**	0.04
andersen	0.36	0.71	2.21	5.28	1.61	0.70	**0.05**	0.10
animals	0.09	0.11	0.46	5.72	0.07	0.18	**0.04**	0.06
buildwall	22.21	13.34	14.04	64.31	27.08	7.04	**0.14**	1.48
cliquer	0.96	0.76	1.70	8.19	2.10	0.33	**0.04**	0.11
downcast	18.58	443.41	90.30	366.32	389.84	42.72	**0.86**	4.35
escape	0.10	**0.03**	0.39	1.76	**0.03**	0.15	**0.03**	0.04
inflammation	0.54	0.13	0.69	2.29	0.11	0.23	**0.04**	0.06
modref	0.17	0.20	0.75	3.34	0.22	0.26	**0.04**	0.05
nearlyscc	19.73	10.87	3.82	37.30	6.02	0.94	**0.07**	0.84
path	0.04	**0.01**	0.37	0.09	**0.01**	0.14	0.03	0.04
polysite	5.09	178.35	48.74	172.95	159.87	20.54	**0.50**	3.53
rsg	0.47	0.57	1.28	4.38	1.22	0.48	**0.04**	0.07
rvcheck	14.89	1.29	0.58	19.63	0.19	0.23	**0.04**	0.08
scc	13.91	8.99	102.51	15.89	7.75	*472.79	**0.15**	47.36
sgen	1.17	12.42	10.61	21.75	10.18	4.00	**0.13**	0.48
ship	0.10	0.10	0.53	2.00	0.07	0.21	**0.04**	0.05
small	0.04	0.05	0.42	1.85	0.05	0.16	**0.03**	0.05
sql-01	0.07	**0.01**	0.37	0.18	**0.01**	0.15	0.03	0.04
sql-02	0.06	**0.01**	0.35	0.19	**0.01**	0.15	0.03	0.03
sql-03	0.35	0.03	0.44	0.38	**0.02**	0.17	0.04	0.04
sql-04	0.04	**0.01**	0.34	0.07	**0.01**	0.14	0.03	0.03
sql-05	0.06	**0.01**	0.35	0.08	**0.01**	0.14	0.03	0.03
sql-06	0.06	**0.01**	0.36	0.11	0.05	0.15	0.03	0.03
sql-07	0.16	0.02	0.41	0.31	**0.01**	0.16	0.03	0.04
sql-08	2.71	0.22	1.22	6.98	0.37	0.37	**0.05**	0.09
sql-09	0.31	**0.02**	0.44	0.97	**0.02**	0.17	0.04	0.04
sql-10	89.85	220.21	28.56	357.57	199.80	14.47	**0.46**	0.89
sql-11	4.19	0.19	*60.82	3.73	0.23	0.26	**0.04**	0.05
sql-12	0.13	**0.03**	TO	0.52	0.04	TO	0.04	0.10
sql-13	0.13	**0.01**	0.39	0.09	**0.01**	0.15	0.03	0.03
sql-14	0.07	**0.02**	0.38	0.20	0.05	0.15	0.03	0.03
sql-15	18.97	39.68	TO	29.39	39.58	TO	**0.77**	TO
traffic	0.07	**0.01**	0.36	0.71	**0.01**	0.15	0.03	0.04
union-find	0.06	0.94	0.75	10.08	0.96	0.33	**0.05**	0.09

7.1 Benchmark Suite

We use ProSynth's benchmark suite, which is descended from the one developed for ALPS [Si et al. 2018] and subsequently used by DiffLog [Si et al. 2019]. It contains files with the expected output tuples for different output relations, and a Soufflé program with the candidate rules generated by ALPS (each one extended with a rule(n) premise). The Soufflé program is compiled into either an executable (for ProSynth) or a shared library (for MonoSynth). For each output relation in the specification, we generate the set of tuples in the complement relation (necessary for MonoSynth and LoopSynth).

The suite consists of 40 benchmarks, including 14 tasks from knowledge discovery, 11 from program analysis, and 15 from relational algebra. The size of the rule sets vary from 5 to 688; for a complete description, see Tables 1 and 2 in the ProSynth paper [Raghothaman et al. 2020].

7.2 MonoSynth

MonoSynth-Z3 achieves ~2× average speedups over ProSynth-Z3 (min/median/geomean/max: 0.03×/2.07×/1.83×/21.94×), and MonoSynth-CVC4 achieves nearly order-of-magnitude average speedups over ProSynth-CVC4 (0.74×/9.08×/9.06×/103.30×). While the CVC4 results indicate a clear improvement, the Z3 results are more ambiguous, as MonoSynth-Z3's performance varies highly across the benchmarks, and most of the speedups over ProSynth-Z3 (20/27) are on benchmarks where ProSynth-Z3 is already fast (i.e., completes in less than a second). We suspect that our heuristics for choosing conflicts might interact badly with Z3's search strategy: ProSynth-Z3 typically makes tens of Datalog calls, and MonoSynth-Z3 makes hundreds. Making so many calls seems to be particularly harmful on the benchmarks where ProSynth-Z3 is much faster (downcast, polysite, and sql-10), as Datalog calls take longer on average on those benchmarks. ProSynth-CVC4 and MonoSynth-CVC4 make about the same number of Datalog calls as MonoSynth-Z3.

All four tools spend most of their time in Datalog solving, and hardly any in SAT solving. This is not surprising, as both ProSynth and MonoSynth in general need to call the Datalog solver more than once per conflict (multiple Datalog calls are required to compute why-not provenance), and the blocking constraints passed to the SAT solver are not particularly complex. Despite the difference in performance relative to their respective baselines (i.e., ProSynth-Z3 and ProSynth-CVC4), both versions of MonoSynth perform the same on average compared to each other.

As noted in Section 4.3, MonoSynth does not need to check for conflicts on every partial assignment. For the experiments, we buffer five assignments to boolean parameters before checking for conflicts. The smaller the buffer, the more quickly we report conflicts, but the more Datalog calls we make (which are costly, despite happening in the same OS process). In our experiments, five seemed to strike a reasonable balance. However, it is far from optimal on all benchmarks, and we anticipate that more sophisticated buffering heuristics, such as choosing the buffer size dynamically based on problem characteristics, could lead to substantial performance improvements.

7.3 LoopSynth

The LoopSynth approaches achieve mixed results that depend on the solver. The results are more positive in the case of CVC4, where LoopSynth achieves solid speedups over ProSynth on average (min/median/geomean/max: 0.00×/6.61×/3.14×/83.87×), while lagging behind MonoSynth (0.00× / 0.34× / 0.35× / 13.81×). Overall, LoopSynth-CVC4 has the fastest time on average out of all the CVC4-backed tools on 11/40 benchmarks. LoopSynth struggles in the case of Z3, in comparison to both ProSynth (0.00×/0.19×/0.24×/25.67×) and MonoSynth (0.00×/0.10×/0.13×/7.71×). Even in this case, LoopSynth-Z3 has the fastest time on average out of the Z3-backed tools on 3/40 benchmarks, and still achieves (occasional) speedups of 26× and 8× over ProSynth-Z3 and MonoSynth-Z3, respectively. The biggest knock against the LoopSynth approach is perhaps its unreliability: the Z3 version times out on 21 trials (across three benchmarks), and the CVC4 version times out on 35 trials (across four benchmarks). On these cases, the algorithm fails to generate loop formulas that force the solver to converge to a stable model in a reasonable time frame.

Compared to ProSynth and MonoSynth, the LoopSynth approach shifts the distribution from time spent in Datalog to time spent in SMT solving. Omitting timed-out results and those taking

Table 3. The ASPSynth approaches encounter fewer conflicts on average than the other approaches; furthermore, conflicts are cheaper, as finding justifications does not require explicit Datalog solving. The definition of a "conflict" varies by tool; this is more of a back-of-envelope calculation than an apples-to-apples comparison. The table excludes timed-out trials.

Tool	# conflicts Min	Median	Arithmean	Max	Definition
ProSynth-Z3	0	8	68	605	# SAT conflicts + # CEGIS iterations - 1
ProSynth-CVC4	0	10	94	807	” ”
MonoSynth-Z3	1	12	83	747	# SAT conflicts + # theory conflicts
MonoSynth-CVC4	1	10	94	707	” ”
LoopSynth-Z3	0	1	122	2420	# SAT conflicts + # loop formulas
LoopSynth-CVC4	0	0	2447	69469	” ”
ASPSynth-Clingo	0	0	52	1418	# conflict lemmas + # loop lemmas
ASPSynth-WASP	0	0	47	1573	# learned clauses + # post-propagators

less than a second, LoopSynth-Z3 spends about an equal percentage of time in Datalog (min/median/arithmetic mean (arithmean)/max: 0.10%/2.47%/3.97%/15.32%) and SMT solving (1.18%/1.83%/2.33%/6.18%); LoopSynth-CVC4 spends a larger percentage of time in SMT solving (7.76%/13.21%/16.15%/31.81%) than in Datalog solving (0.27%/2.90%/9.80%/28.96%). Much of the rest of the time is spent in encoding the Clark completion and constructing loop formulas.

Fifteen of the 40 benchmarks are tight, and consequently require no loop formulas. On the remaining benchmarks (that do not time out), LoopSynth-Z3 and LoopSynth-CVC4 generate only a handful of loop formulas in the median, but can generate hundreds or thousands in bad cases (min/median/arithmean/max: 0.00/2.00/28.99/442.30 and 0.00/1.00/374.51/7241.00, respectively). Each loop formula results in one Datalog call; the relatively low number of Datalog calls softens the disadvantage that LoopSynth uses Gringo to interpret the Datalog programs, instead of invoking a more efficient compiled version of the program generated by Soufflé (like ProSynth and MonoSynth).

7.4 ASPSynth

The direct ASP encodings are the most effective solution on the benchmark suite. ASPSynth-Clingo completes every trial in less than one second, is the fastest tool on average on 26/40 benchmarks, and achieves on average ~9× speedups over ProSynth-Z3 (min/median/geomean/max: 1.16×/7.50×/9.47×/372.19×). ASPSynth-WASP also has solid performance, with speedups of ~3× over ProSynth-Z3 (0.03×/2.63×/3.62×/186.10×). However, it consistently times out on the benchmark sql-15. Intriguingly, both LoopSynth tools do the same: there may be something in the benchmark's cyclic structure that is difficult for some ASP-based approaches to tame.

What makes the direct-to-ASP tools more effective than the other approaches? One factor—that should not be discounted—is that Clingo and WASP are sophisticated, highly engineered tools that have been improved over years, whereas the other approaches are ad hoc and not highly optimized. Clingo and WASP avoid some overheads faced by the other implementations, such as multiple OS processes, or even in-memory translation between the representations of different systems (e.g., Z3 and Soufflé). Similarly, ASPSynth does not need to perform Horn clause evaluation multiple times: it grounds the program just once, whereas the other tools run an explicit Datalog evaluation to check whether each proposed solution satisfies the least model semantics.

Furthermore, the tight integration of Horn clause solving and SAT search within an ASP solver means that it can more effectively search the solution space. To get a sense for this, we can look at the number of formulas that each approach adds to the SAT solver to guide it to a solution; in general, the higher the number, the more unproductive space the SAT solver has explored. For concision, we refer to these formulas as "conflicts"; the precise definition varies by tool, and a comparison between them should not be taken as apples-to-apples (e.g., a single loop formula generated by LoopSynth might translate to multiple clauses in the SAT solver; we undercount the number of conflicts encountered by ProSynth, since a single CEGIS iteration might encounter multiple conflicts). Nonetheless, they provide some picture into the performance of the different algorithms (Table 3). The ProSynth and MonoSynth approaches encounter about the same number of conflicts. The LoopSynth approaches encounter one or fewer conflicts in the median, but can also encounter many in bad cases. The direct ASP encodings have, on average, the fewest conflicts, with a median of zero conflicts. Furthermore, not only do they encounter fewer conflicts, but it is also more efficient for them to construct conflict justifications, since—unlike ProSynth or MonoSynth—they do not need to invoke a Datalog solver to generate provenance. Thus, the direct ASP encodings have a winning combination of fewer conflicts that are cheaper to compute.

7.5 Bigger Picture: How Effective is ASPSynth-Clingo?

One contribution of our work is an exploration of the design space around solver-backed algorithms for solving Datalog synthesis-as-rule-selection problems. That being said, now that we have a dominant solution—ASPSynth-Clingo—we might wonder how it stacks up against other possible techniques out there. This section further evaluates how effective a tool ASPSynth-Clingo is for solving Datalog synthesis-as-rule-selection problems. Overall, we find that it is a very effective tool on the ProSynth benchmark suite and scales well compared to other approaches based on candidate rule sets. We also suggest some ways to overcome current limitations.

7.5.1 Contestants. Solution quality matters in program synthesis. We evaluate two versions of ASPSynth-Clingo: the original (doing no solution minimization) and ASPSynth-Clingo-MinPremise, which minimizes the total number of premises in the solution (following Section 6.1.2). We also compare to GenSynth [Mendelson et al. 2021], an ILASP2 [Law et al. 2015] encoding, and ProSynth.

GenSynth [Mendelson et al. 2021] is a Datalog synthesis tool that uses a genetic programming algorithm to produce candidate rules instead of selecting them from a pre-existing set. Compared to the other approaches, which select from a candidate rule set, this can be both a disadvantage (when the candidate rule set is small) or an advantage (when the candidate rule set is large or hard to filter). The GenSynth paper reports speedups over a version of ProSynth that uses Soufflé in an interpreted mode (normally ProSynth interacts with executables generated by Soufflé); the GenSynth evaluation did not include a comparison against ProSynth using compiled Soufflé programs. GenSynth itself uses Soufflé in an interpreted mode, and attempts to minimize solutions.

Using a meta-level approach, ILASP2 [Law et al. 2015] encodes ASP program synthesis as ASP programs that it discharges to Clingo. Because Datalog is a fragment of ASP, we can naturally encode Datalog synthesis in ILASP2, similarly to the direct ASP encoding of Section 6 (ILASP allows candidate rules to be explicitly enumerated, so we omit the rule(n) atoms and choice rules). When configuring ILASP2, we set the upper bound for the number of literals in the synthesized program to 30, which is twice the default value (the minimum value that works is 28). Compared to ASPSynth, ILASP2 generates more complex ASP programs (involving meta-level machinery), and makes two calls to Clingo (instead of one). ILASP2 minimizes the number of premises in solutions.

As a baseline, we use a lightly modified version of the ProSynth implementation of Raghothaman et al. [2020] (we turn off logging and record additional statistics). The original ProSynth paper does not report the time it takes Soufflé to compile the candidate rule sets (compilation happens only once per rule set; the resulting executable is invoked multiple times). To focus on the solver-based aspects of the approaches, we also omitted this time from the ProSynth and MonoSynth numbers in the previous section. However, here we report ProSynth as execution time with and without compilation time, on the basis that—if one were to actually want to use ProSynth to synthesize a program from scratch—it would be necessary to compile the candidate rules.[5]

7.5.2 Benchmarks. We use two sets of benchmarks. The first consists of the 40 ProSynth benchmarks discussed in Section 7.1. The second is a set of scaling benchmarks that vary the number of candidate rules and the size of the specification. Each of these benchmarks tries to synthesize a program for computing strongly connected components. For candidate rules, we consider 100 rules, 500 rules, and 1000 rules (taken from the scaling experiment in the ProSynth paper [Raghothaman et al. 2020]). For specification size, we consider EDBs consisting of 10 tuples (scc-1x), 100 tuples (scc-10x), and 1000 tuples (scc-100x) (taken from the scaling experiment in the GenSynth paper [Mendelson et al. 2021]). We consider each combination of rule set size and specification size.

7.5.3 Results. ASPSynth-Clingo is, on average, substantially faster on the ProSynth benchmark suite than other approaches (Figure 10(a)). It is significantly faster than ProSynth without counting Soufflé compilation time (min/median/geometric mean/max speedup: 0.65×/7.58×/9.33×/828.86×) and orders of magnitude faster than GenSynth (6.88× / 140.58× / 200.16× / 20000×). The ILASP2 encoding is also significantly faster than ProSynth (0.21×/2.82×/3.48×/236.14×) and GenSynth (4.29×/45.01×/74.57×/6593.40×), a further indication of the suitability of ASP for these Datalog synthesis tasks. ASPSynth-Clingo is faster than ILASP2 (1.31× / 2.89× / 2.68× / 5.07×). Premise minimization does not cost ASPSynth-Clingo much (speedup over -MinPremise: median 1.00×, geometric mean 1.06×). GenSynth produces the minimal output, counting either rules or premises. GenSynth's solutions have moderately fewer premises than ASPSynth-Clingo-MinPremise and ILASP2 (median 1.00×, geomean 0.86×), and substantially fewer premises than ASPSynth-Clingo (median 0.67×, geomean 0.54×) and ProSynth (median 0.67×, geomean 0.56×).

GenSynth scales best as we vary the number of candidate rules and the number of example input/output tuples (Figure 10(b)). It is relatively consistent across the configurations (but for a timeout in scc-1x); it beats ProSynth in all but the smallest configuration. The ASP-based approaches scale well if only *one* dimension is scaled up, but fall behind GenSynth when *both* dimensions are scaled simultaneously. The largest configurations highlight the differences in performance between ASPSynth-Clingo, ASPSynth-Clingo-minpremise, and ILASP2. While GenSynth scales well, it is the slowest on the benchmark suite (Figure 10(a)).

8 DISCUSSION

We discuss ASPSynth limitations (Section 8.1), the difficulties of comparing solver-backed algorithms (Section 8.2), critiques of the rule selection problem (Section 8.3), and a wider view of it (Section 8.4).

8.1 Limitations of ASPSynth

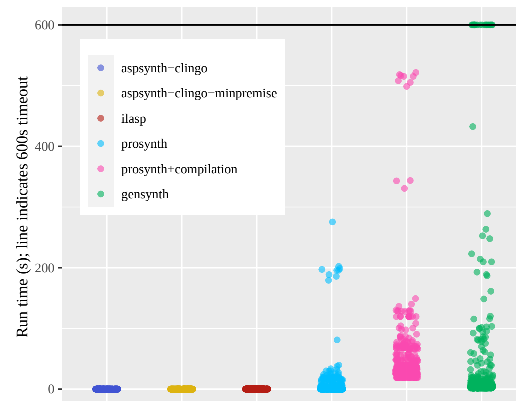
While ASPSynth outperforms other solutions, it still suffers from several limitations.

Grounding is typically a major bottleneck for ASP-based logic programming synthesis tools [Athakravi et al. 2013; Cropper and Morel 2021]. ASPSynth's performance degrades with the size of the specification, due to a combinatorial explosion in the number of ground rules; it would benefit from advances in ASP solvers that combine lazy grounding and SAT techniques [Weinzierl 2017].

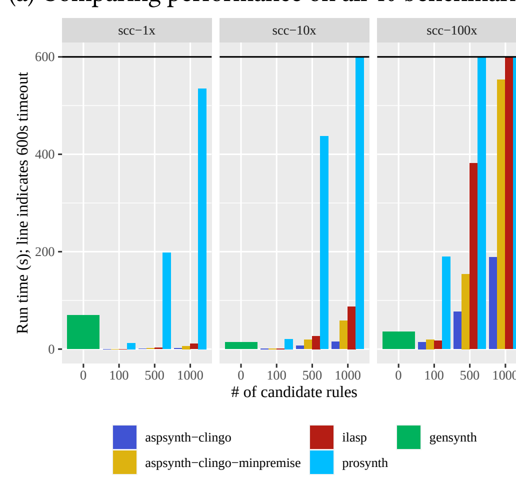
[5] We could, alternatively, evaluate ProSynth using Soufflé in interpreted mode, as was done in the GenSynth paper evaluation [Mendelson et al. 2021]. Both our informal experience and the numbers reported by GenSynth suggest that this mode is very slow (potentially slower than compiling the programs with Soufflé and then running the normal version of ProSynth).

Our approaches do not handle noisy specifications (where $\mathcal{T}^+_{exp}$ or $\mathcal{T}^-_{exp}$ contains spurious tuples). ASPSynth could easily support them: replace hard constraints (C^+) and (C^-) with soft constraints penalizing missed positive and negative examples (following RASPAL [Athakravi et al. 2013]).

ASPSynth could be extended to synthesize programs with stratified negation [Apt et al. 1988; Przymusinski 1988; Van Gelder 1989], a common Datalog approach to negation that disallows predicates defined (directly or transitively) by their own negation. Synthesizing programs with negation is itself not a problem (since our encoding can synthesize general ASP programs). To ensure we synthesize only stratified programs, we could add ASP rules that define when there is a negative dependency between predicates and add hard constraints to prevent each predicate from negatively depending on itself.



(a) Comparing performance on all 40 benchmarks.



(b) Comparing as inputs scale.

Fig. 10. ASP performs well compared to other approaches. (a) Across 40 benchmarks, ASP-based encodings (ASPSynth-Clingo and ILASP) are, on average, substantially faster than ProSynth (median 7.58× and 2.82×) and orders of magnitude faster than GenSynth (median 141× and 45×). (b) ASP-based approaches scale better than ProSynth, and outperform GenSynth when varying *either* the number of candidate rules *or* the size of the specification. However, they fall behind GenSynth when both dimensions are scaled up.

8.2 Evaluating Solver-Backed Algorithms

We have presented a progression of solver-backed algorithms for solving the Datalog synthesis-as-rule-selection problem. For each approach, we implemented two versions, each using a different backend solver, and then evaluated these implementations on a shared benchmark suite. The intention behind using multiple backend solvers was to reduce the noise of idiosyncratic solver behavior, and thus make more general claims about the effectiveness of each algorithm.

In essence, we would like to be able to make a statement about the effectiveness of solver-backed *algorithms*, and not about particular solver-backed *tools*. However, it is in general very hard to evaluate a solver-backed algorithm and not a particular implementation of it. Different solver-backed approaches are hard to distinguish from a complexity perspective—the problems they solve are often intractable. An asymptotic analysis is not usually informative—and even when it is, practical performance does not always align with theoretical performance. Furthermore, solvers are complex, blackbox tools with sometimes unpredictable and hard-to-explain behavior; thus, an empirical measure of an implementation can also fail to shed light on the characteristics of the algorithm itself. For example, ProSynth-Z3 performs much better than ProSynth-CVC4. Which version gives a "better" picture of the performance of ProSynth—the algorithm, not the tool—relative to, say, MonoSynth the algorithm (note that MonoSynth-Z3 and MonoSynth-CVC4 perform very similarly)? Is it possible to evaluate a solver-backed algorithm in the abstract, without reference to the idiosyncrasies of whatever concrete solver is chosen as the backend?

8.3 Critiques of the Synthesis Problem

There are two clear applications for synthesizing Datalog rules that match a specification; however, neither is a good match for the current framing of the Datalog synthesis-as-rule-selection problem.

Synthesizing Datalog Programs. The goal is to synthesize Datalog code as an alternative (or a supplement) to handwriting Datalog programs. With its focus on programs that might normally be written by hand (e.g., static analyses, SQL queries), the ProSynth benchmark suite reflects this traditional PL view. Given a proposed Datalog synthesis tool, the key question is, "Can this tool be used to synthesize programs of interest?" The current benchmark suite is not entirely satisfactory by this measure, as many benchmarks are trivial to write by hand (like scc). More fundamentally, the current framing of the problem is problematic. By framing Datalog synthesis as a *filtering* problem [Markovitch and Scott 1993], it avoids a key question that is especially knotty in this context: where do candidate rules come from? Datalog synthesis-as-rule-selection relies on the assumption that candidate rules can be explicitly enumerated, an unreasonable assumption in certain scenarios. Furthermore, from a practical perspective, the set of candidate rules has to be relatively small (though ASP-based approaches scale to larger sets than ProSynth; Figure 10(b)). These candidate rules are a form of bias, which is critical for learning meaningful programs [Cropper et al. 2021]; yet it is nontrivial to generate candidate rules for realistic Datalog programs in a way that avoids substantial programmer input. For example, the benchmark suite uses rules produced by ALPS' meta-rule-based approach [Si et al. 2018]; many of the program analysis benchmarks need specialized meta-rules, which presumably requires domain expertise and tuning. What's more, given a candidate rule set for a complex program, it might be very difficult to choose an input-output example that is strong enough to lead to an actual solution: even on a program as simple as scc, the specification in the benchmark suite is not sufficient to guarantee a correct rule selection (i.e., one that generalizes to all graphs).

Logic-Based Machine Learning. From an artificial intelligence perspective, one might synthesize Datalog rules as a way to explain how input data leads to output data, i.e., logic-based machine learning [Cropper et al. 2021]. It is conceivable that one could generate candidate rules in this setting, if possible explanations tend to follow patterns. Nonetheless, the current framing of Datalog synthesis-as-rule-selection still feels limiting in this situation. First, given the nature of machine learning, it would seem necessary to handle noise, when tuples are incorrectly marked as expected or unexpected. Second, it is too restrictive to frame the problem in terms of a single input-output example, instead of multiple ones. Practically, the benchmark suite would have to be extended with benchmarks that more accurately represent this setting (e.g., large numbers of expected and unexpected tuples).

In both settings, it is important to somehow measure or characterize how good solutions are. After all, a Datalog synthesis tool could always use the output tuples as the synthesized rules themselves—a perfect, unbeatably fast solution every time! Such an approach is risible, of course: it pathologically overfits to the example. Ideally, the framing of the synthesis problem would encourage generalizable solutions. Biasing towards small solutions is important for human understanding: a user of a synthesis tool will inspect the selected candidate rules and rename the variables to have semantically meaningful names. Small solutions may also generalize better/overfit less, a particular concern considering that the conventional framing of Datalog synthesis-as-rule-selection uses only a single example. If we relax the synthesis problem to allow for multiple, noisy examples, then it should be possible to separate our examples into training and test data. Given enough examples, we could use cross-validation to measure how well a given solution generalizes.

Furthermore, without a clear idea of the target application, it is hard to evaluate which assumptions and experimental setups make sense. For example, the ProSynth paper [Raghothaman et al. 2020] uses Soufflé in compiled mode but excludes compilation time from the benchmarks. In contrast, the GenSynth paper [Mendelson et al. 2021] compares against ProSynth using Soufflé in interpreted mode, which it bests.[6] It is important that tools be compared in an apples-to-apples way, i.e., best configuration vs best configuration, counting time appropriately. Admittedly, what is "appropriate" depends on the context. In a setting where the candidate rules do not change but the data does (e.g., ML), it might be reasonable to omit the one-time cost of compilation. In a setting where the candidate rules are specific to each problem (like synthesizing a particular static analysis), then it seems important to include compilation time (and candidate rule generation time). Relatedly, without an idea of the context and clients, it is difficult to evaluate whether the speed of a Datalog synthesis tool is reasonable. If the tool is meant to speed up program development, then it is appealing only if the time to 1) come up with an example and candidate rules, 2) synthesize the program, and 3) inspect and approve it is substantially less than the time to program it and test it. Our timeout of 600 seconds—ten minutes—is quite generous for an interactive setting; however, it might be short for an ML setting. Future work should justify its timeout based on who will use the synthesis tool and how.

8.4 Beyond Datalog Synthesis

We have critiqued the Datalog synthesis-as-rule-selection problem; this section offers a more optimistic view. ProSynth is a tool for solving an apparently limited synthesis problem; however, at an implementation level, it does this by selecting rule(n) facts in the EDB. This suggests that ProSynth could easily be extended to be a much more general tool: given a Datalog program P, a specification, and a universe of potential EDB facts, choose a subset of those facts I such that $P(I)$ meets the specification. If the specification defines, say, the negation of a safety property, this amounts to a form of bounded model checking for Datalog: given a bounded universe of potential facts, is there any combination of facts that can lead to a violation of the safety property? For example, one could compose a program consisting of two access control policies and ask, "Given this universe of facts, can any combination of facts lead to one policy granting a resource while the other denies it?" This new framing would weaken some of our critiques of the previous problem (e.g., longer runtimes is less of an issue for model checking) and shift others (e.g., instead of having to come up with candidate rules, you now need to come up with a universe of candidate facts).

All the approaches we propose in this paper could be used to solve this wider problem. The benchmark problems we use in Section 7 are all synthesis problems; we would need to devise new benchmarks to see whether our approaches work well for this task. Generally speaking, ASP seems to be a promising tool for reasoning about Datalog programs, as it combines SAT search with Datalog evaluation. The grounding stage of ASP solving might be prohibitive for full scale Datalog programs (like a Java points-to analysis [Bravenboer and Smaragdakis 2009]); on the other hand, given the similarities between grounding and Datalog evaluation, it is conceivable that techniques

[6] Excluding compilation time, ProSynth performs better than GenSynth on the benchmark suite. GenSynth cannot reasonably use Soufflé in compiled mode, as it generates many Soufflé programs and Soufflé compilation times are relatively high.

that have helped Datalog scale (such as parallelism and compilation) could also help grounders scale to larger ASP problems.

9 RELATED WORK

Datalog Synthesis. Zaatar encodes Datalog synthesis problems into SMT constraints, including constraints that restrict the form that rules can take [Albarghouthi et al. 2017]. ALPS generates candidate rules from meta-rules; it selects from the candidate rule set using a bidirectional search strategy [Si et al. 2018]. DiffLog attaches real values to candidate rules and then uses a technique inspired by numerical relaxation to choose among them [Si et al. 2019]. ProSynth outperforms both ALPS and DiffLog. GenSynth uses an evolutionary algorithm to generate rules; it can also handle noisy specifications [Mendelson et al. 2021]. The Apperception Engine synthesizes causal theories for infinite sequences of sensory inputs by encoding an interpreter for a causal variant of Datalog in ASP [Evans et al. 2021]. Although their setting does not quite match ours, they show that their approach produces much smaller ground ASP programs than ILASP; hence, it might be a promising direction forward for avoiding the grounding bottleneck. Datalog synthesis is related to relational query synthesis, a recent example of which is EGS [Thakkar et al. 2021]; however, relational query synthesis usually does not involve synthesizing recursion. See the ProSynth paper for a survey [Raghothaman et al. 2020].

Inductive Logic Programming (ILP). ILP is the field of synthesizing logic programs, given some background information (in the form of a logic program) and examples [Cropper et al. 2021; Muggleton 1991]. Many ILP systems target Prolog. While Datalog is a syntactic subset of Prolog, the two have different evaluation strategies and are typically used for different tasks. It is unlikely that tools optimized for Prolog synthesis would be optimal for Datalog synthesis. We evaluate ILASP [Law et al. 2020a] in Section 7.5 because it targets ASP, a better match for Datalog.

ASPSynth is closest to ILP systems that use ASP [Kaminski et al. 2018; Law et al. 2020a; Schüller and Benz 2018]. ASPAL iteratively builds a hypothesis space of skeleton ASP rules [Corapi et al. 2011]; at each iteration, it discharges rule instantiation to an ASP solver using a similar encoding to our direct one (Section 6). RASPAL improves upon ASPAL by refining the hypothesis space within iterations, leading to smaller ground programs [Athakravi et al. 2013]. Popper also avoids posing large ASP queries by iteratively exploring the hypothesis space [Cropper and Morel 2021]; however, it does not allow candidate rules to be explicitly enumerated (similar to GenSynth [Mendelson et al. 2021]), and preliminary experiments using v1.0.2 were not competitive on the ProSynth benchmark suite. FastLAS generates an optimized set of candidate ASP rules and then chooses among them using a single solver call [Law et al. 2020b, 2021], but does not support recursive rules.

ILP systems, including ASP-based ones, typically provide a much richer feature set than our ASPSynth encoding. For example, ILASP [Law et al. 2020a] provides multiple ways to specify language bias, which determines what is known in ILP as the hypothesis space—the space of candidate programs considered during synthesis. ASPSynth expects candidate rules to be enumerated explicitly; ILASP supports this (the encoding we use in Section 7.5), while providing three additional ways to specify the hypothesis space: mode declarations (specifying the shape of predicates and how they can be used in rules), meta-rules (like ALPS [Si et al. 2018]), and even ASP programs (a meta-level approach). By supporting multiple, potentially noisy examples, ILASP contemplates the more complex synthesis settings that we argue Datalog synthesis should consider (Section 8.3).

ASPSynth's approach to rule selection is also similar to techniques used to translate between different flavors of probabilistic logic programming [Balai and Gelfond 2016; Lee et al. 2017].

Combining Solvers and Horn Clause Evaluation. Some prior work augments logic programming with the ability to interact with an external solver, such as the framework of constraint logic programming [Jaffar and Lassez 1987; Jaffar and Maher 1994], and ad hoc systems like Calypso [Aiken et al. 2007; Hackett 2010] and Formulog [Bembenek et al. 2020]. Other approaches embed Horn clause solving within systems that perform symbolic reasoning. Constrained Horn clause solvers [Bjørner et al. 2015; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder and Bjørner 2012] find symbolic solutions to systems of Horn clauses (e.g., given a predicate p that appears in a system of clauses, they might determine that $p(x, y)$ holds iff $x < 2y$). The fixed point Horn clause solver μZ [Hoder et al. 2011] is embedded inside the SMT solver Z3; it supports Datalog evaluation with some symbolic reasoning. However, unlike the theory of Datalog discussed in Section 4, Horn clause solving seems to be separate from satisfiability checking: in an environment where the rules {`p :- q.`, `q.`} have been asserted, it returns that the formula ¬p is satisfiable.

The ASP solver Cmodels2 [Giunchiglia et al. 2004] refines ASSAT by a closer integration with the SAT solver and produces clauses that are simpler than loop formulas (but weaker). Our implementation follows the closer integration with the SAT solver, but produces ASSAT-style loop formulas, which we found to be more effective. Gebser et al. [2014] solve ASP problems using a SAT solver extended with a theory of acyclic graphs. Since it could be used to solve general SAT problems involving Datalog predicates, it is an alternative to our monotonic theory and ASSAT-based approaches; it is closer to the latter (it encodes the ground program as a SAT formula and then checks whether partial models meet acyclicity requirements). Constraint answer set programming (CASP) [Gebser et al. 2009; Mellarkod et al. 2008] extends answer set programming with predicates representing constraints from different theories. Our monotonic theory and ASSAT-based approaches support solving SMT formulas including Datalog predicates; conversely, CASP makes it possible to solve Horn clauses containing theory predicates.

10 OUTLOOK

In exploring approaches to the Datalog synthesis-as-rule-selection problem, we have found that the *simplest* approach—a straightforward encoding into ASP, easily solved by existing tools—is also the most *performant* approach, leading to ~9× geomean speedups over the prior state of the art. Along the way to this solution, we proposed two techniques that can be used to solve arbitrary SMT formulas containing Datalog predicates, and showed that it is possible to construct (theoretically) efficient monotonic SMT theories from Datalog programs. We identified shortcomings with the existing framing of the Datalog synthesis-as-rule-selection problem and proposed that the problem could be generalized to a form of bounded model checking of Datalog programs.

Given its usefulness in solving a problem of interest to the programming language community, we suggest that ASP is a promising technique that is underutilized in PL circles. ASP is very effective at solving logical search tasks involving fixed points, which indicates that it could be a powerful tool for reasoning about Datalog programs (although it is unclear if ASP scales to reasoning about larger Datalog programs, like a realistic Java points-to analysis). Furthermore, while ASP solving might not prove to be as general a tool as SMT solving, we anticipate that the PL community could profitably use ASP to solve problems outside the context of Datalog.

We plan to explore three main directions in future work. First, we would like to investigate more realistic Datalog synthesis problems where it might be feasible to generate a candidate rule set by mutating the rules of an existing Datalog program. Second, we would like to identify additional use cases for Datalog-based monotonic theories, and improve our theory implementations to scale to larger problems. Third, we plan to explore the application of ASP solving to PL problems beyond Datalog synthesis, such as synthesizing programs in other query languages.

ACKNOWLEDGMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) Contract No. FA8750-19-C-0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). Emery Berger gave valuable presentation advice. Noam Ross helped with the charts, and Alice Feng helped with the narrative. Aaditya Naik, Aalok Thakkur, and Mukund Raghothaman answered our questions about ProSynth, GenSynth, and the benchmark suite, and Mark Law answered our questions about ILASP2. We thank Nada Amin, Andrew Cropper, Mayur Naik, and Bernhard Scholz for their thoughtful feedback on drafts of this paper. We also thank Nada Amin for introducing us to truth maintenance systems.

REFERENCES

Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 43–48. https://doi.org/10.1145/1251535.1251543

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*. 689–706.

Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Hellerstein, and Russell Sears. 2010a. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European Conference on Computer systems*. 223–236.

Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2010b. Dedalus: Datalog in Time and Space. In *Datalog 2.0*. 43–48.

Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. 2013. WASP: A Native ASP Solver Based on Constraint Learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning*. 54–66. https://doi.org/10.1007/978-3-642-40564-8_6

Krzysztof R Apt, Howard A Blair, and Adrian Walker. 1988. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*. Elsevier, 89–148. https://doi.org/10.1016/B978-0-934613-40-8.50006-3

Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. 2013. Learning Through Hypothesis Refinement Using Answer Set Programming. In *Proceedings of the 23rd Interational Conference on Inductive Logic Programming*. 31–46.

John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark A. Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. Reachability Analysis for AWS-Based Networks. In *Proceedings of the 31st International Conference on Computer Aided Verification*. 231–241.

Evgenii Balai and Michael Gelfond. 2016. On the Relationship Between P-log and $\mathrm{LP}^{\mathrm{MLN}}$. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. 915–921.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT Modulo Monotonic Theories. In *Proceedings of the 29th Conference on Artificial Intelligence*. 3702–3709.

Sam Bayless, Nodir Kodirov, Syed M. Iqbal, Ivan Beschastnikh, Holger H. Hoos, and Alan J. Hu. 2020. Scalable Constraint-Based Virtual Data Center Allocation. *Artif. Intell.* 278 (2020), 103196.

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 141:1–141:31. https://doi.org/10.1145/3428209

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II*. Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 243–262.

Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. 2011. Answer Set Programming at a Glance. *Commun. ACM* (2011). https://doi.org/10.1145/2043174.2043195

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory*. 316–330.

Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. 2017. I-DLV: The New Intelligent Grounder of DLV. *Intelligenza Artificiale* 11, 1 (2017), 5–20.

Keith L Clark. 1977. Negation as Failure. In *Logic and Data Bases*. 293–322.

Domenico Corapi, Alessandra Russo, and Emil Lupu. 2011. Inductive Logic Programming in Answer Set Programming. In *Proceedings of the 21st International Conference on Inductive Logic Programming*. 91–97.

Andrew Cropper, Sebastijan DumanÄŊiÄĞ, Richard Evans, and Stephen H. Muggleton. 2021. Inductive Logic Programming at 30. *Mach. Learn.* (2021).

Andrew Cropper and Rolf Morel. 2021. Learning Programs by Learning From Failures. *Mach. Learn.* (2021).

Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. 632–646.

Jon Doyle. 1979. A truth maintenance system. *Artif. Intell.* 12, 3 (1979), 231–272.

Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek J. Sergot. 2021. Making Sense of Sensory Input. *Artif. Intell.* (2021), 103438.

Francois Fages. 1994. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science* 1, 1 (1994), 51–60.

Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *Proceedings of the 29th USENIX Security Symposium*. 1075–1092.

Hervé Gallaire and Jack Minker (Eds.). 1978. *Logic and Data Bases*. Plenum Press.

Martin Gebser, Tomi Janhunen, and Jussi Rintanen. 2014. Answer Set Programming as SAT Modulo Acyclicity. In *Proceedings of the 21st European Conference on Artificial Intelligence*. 351–356.

Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. 2011a. Advances in Gringo Series 3. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning*. 345–351.

Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011b. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* (2011).

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Conflict-Driven Answer Set Solving: From Theory to Practice. *Artif. Intell.* (2012), 52–89.

Martin Gebser, Max Ostrowski, and Torsten Schaub. 2009. Constraint Answer Set Solving. In *Proceedings of the 25th International Conference on Logic Programming*. 235–249.

Martin Gebser, Torsten Schaub, and Sven Thiele. 2007. GrinGo: A New Grounder for Answer Set Programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*. 266–271.

Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. 2004. SAT-Based Answer Set Programming. In *Proceedings of the 19th National Conference on Artificial Intelligence*. 61–66.

Sergey Grebenshchikov, Nuno Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 405–416. https://doi.org/10.1145/2254064.2254112

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering*. 1176–1186.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 116:1–116:27.

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Foundations & Trends in Databases* (2013).

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

Salvatore Guarnieri and V. Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of the 18th USENIX Security Symposium*. 78–85.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*. 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

Brian Hackett. 2010. *Type Safety in the Linux Kernel*. Ph. D. Dissertation. Stanford University.

Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. 2009. Artemis: A System for Analyzing Missing Answers. *Proc. VLDB Endow.* (2009).

Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–171. https://doi.org/10.1007/978-3-642-31612-8_13

Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ–An Efficient Engine for Fixed Points with Constraints. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. 457–462.

Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 111–119. https://doi.org/10.1145/41625.41635

Joxan Jaffar and Michael J. Maher. 1994. Constraint Logic Programming: A Survey. *The Journal of Logic Programming* 19 (1994), 503–581. https://doi.org/10.1016/0743-1066(94)90033-7

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Proceedings of the 28th International Conference on Computer Aided Verification*.

Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. 2018. Exploiting Answer Set Programming With External Sources for Meta-Interpretive Learning. *Theory Pract. Log. Program.* 18, 3-4 (2018), 571–588.

Tobias Klenze, Sam Bayless, and Alan J. Hu. 2016. Fast, Flexible, and Minimal CTL Synthesis via SMT. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 136–156.

Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. 2020b. FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria. In *Proceedings of the 34th Conference on Artificial Intelligence*. 2877–2885.

Mark Law, Alessandra Russo, and Krysia Broda. 2015. Learning Weak Constraints in Answer Set Programming. *Theory Pract. Log. Program.* 15, 4-5 (2015), 511–525.

Mark Law, Alessandra Russo, and Krysia Broda. 2020a. The ILASP System for Inductive Learning of Answer Set Programs. *ALP Newsletter* (2020).

Mark Law, Alessandra Russo, Krysia Broda, and Elisa Bertino. 2021. Scalable Non-observational Predicate Learning in ASP. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*. 1936–1943.

Joohyung Lee, Samidh Talsania, and Yi Wang. 2017. Computing $\mathrm{LP}^{\mathrm{MLN}}$ Using ASP and MLN Solvers. *Theory Pract. Log. Program.* 17, 5-6 (2017), 942–960.

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

Ninghui Li and John C Mitchell. 2003. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*. 58–73.

Fangzhen Lin and Yuting Zhao. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artif. Intell.* 157, 1-2 (2004), 115–137.

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*. 271–286.

Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 97–108.

Shaul Markovitch and Paul D. Scott. 1993. Information Filtering: Selection Mechanisms in Learning Systems. *Mach. Learn.* 10, 2 (1993), 113–151. https://doi.org/10.1007/BF00993503

David A McAllester. 1990. Truth Maintenance. In *Proceedings of the 8th National Conference on Artificial Intelligence*. 1109–1116.

Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. 2008. Integrating Answer Set Programming and Constraint Logic Programming. *Ann. Math. Artif. Intell.* 53, 1 (2008), 251–287.

Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GenSynth: Synthesizing Datalog Programs without Language Bias. In *Proceedings of the 35th Conference on Artificial Intelligence*. 6444–6453.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

Stephen Muggleton. 1991. Inductive Logic Programming. *New Gener. Comput.* 8, 4 (1991), 295–318.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (2006), 937–977.

C. H. Papadimitriou. 1985. A note on the Expressive Power of Prolog. *Bull. EATCS* 26 (1985), 21–22.

Teodor C Przymusinski. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*. Elsevier, 193–216. https://doi.org/10.1016/b978-0-934613-40-8.50009-9

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-Guided Synthesis of Datalog Programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62:1–62:27.

Thomas W. Reps. 1995. Demand Interprocedural Program Analysis Using Logic Databases. In *Applications of Logic Databases*.

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry*. 56–67.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.

Peter Schüller and Mishal Benz. 2018. Best-Effort Inductive Logic Programming via Fine-Grained Cost-Based Hypothesis Generation. *Mach. Learn.* 107, 7 (2018), 1141–1169.

Roberto Sebastiani. 2007. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3, 3-4 (2007), 141–224.

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineeering Conference and Symposium on the Foundations of Software Engineering*. 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*. 6117–6124.

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*. 404–415.

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1110–1125.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

Allen Van Gelder. 1989. Negation as Failure Using Tight Derivations for General Logic Programs. *The Journal of Logic Programming* 6, 1-2 (1989), 109–133. https://doi.org/10.1016/0743-1066(89)90032-0

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM* 38, 3 (1991), 619–649.

Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*. 137–146.

Antonius Weinzierl. 2017. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning*. 191–204.

John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. 131–144.

Allison Woodruff and Michael Stonebraker. 1997. Supporting Fine-Grained Data Lineage in a Database Visualization Environment. In *Proceedings of the Thirteenth International Conference on Data Engineering*. 91–102.

David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 7:1–7:35.