From Program Analyses to Transformations with Datalog

Manish Shetty
UC Berkeley, USA
manishs@berkeley.edu

Abstract

Datalog has proven to be a powerful language for encoding program analyses, but its connection to program transformations remains underexplored. This paper investigates how ideas from database repair, data provenance, and symbolic execution can bridge the gap between using Datalog for program analysis and leveraging it for program transformations. We survey existing approaches and present an upcoming perspective on the symbolic execution of Datalog rules, which simultaneously enables the representation of a space of potential input database (program) changes and computing their effect on the analysis results. We posit that bridging Datalog's strengths for analysis with an ability to reason about and execute transformations would significantly expand its utility for program optimization, repair, and other transformations.

1 Introduction

The use of deductive databases and logic programming languages, such as Datalog (Abiteboul et al., 1995), for program analysis is far from new (Reps, 1995; Whaley & Lam, 2004; Lam et al., 2005). The declarativeness of Datalog makes it attractive for specifying complex analyses (Jordan et al., 2016; Grech et al., 2018). The ability to specify recursive definitions is particularly exciting, as program analysis is fundamentally a mixture of mutually recursive tasks (Figure 1).

A Datalog query is a set of Horn clauses such as `path(X,Y) :- edge(X,Z), path(Z,Y).` executed against a database of facts referred to as the extensional database (EDB). The result is a set of derived facts, referred to as the intensional database (IDB).

Naturally, a program to be analyzed is represented as an EDB; e.g., control-flow graph (CFG) as a set of facts. When a query is executed on this EDB, an optimized Datalog engine computes all facts that can be inferred using the query's rules. A big advantage of this setup is the multitude of optimizations from the classical database systems that come for free to make these analyses efficient (Bravenboer & Smaragdakis, 2009).

While connecting program analysis to databases (viā Datalog) has proved useful, an open question is whether *program transformations similarly translate to useful aspects of databases*.

Program transformations involve changes in a program to improve an objective. Reasoning about these changes is necessary in many practical applications. For example, while a program analysis can determine if a program's behavior violates a certain property, a valid repair (fix, exception handler, etc.) ensures a program that meets the desired property Sadowski et al. (2018). Consequently, reflecting on what transformations lead to a desired property is of great interest.

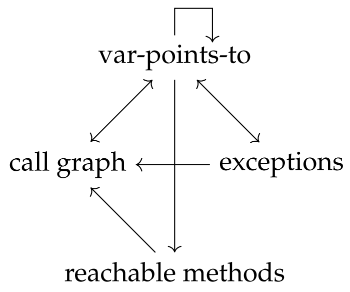


Figure 1: A subset of program analyses illustrating a domain of mutual recursion

2 Existing Connections

Starting from a datalog view of programs represented as an EDB, we naturally arrive at program transformations being transformations to the corresponding facts in the database. Consequently, we can view the challenges of reflecting on these database transformations modulo a desired property in two phases. First, is the *reflection* aspect that answers what is the origin or derivation of an inferred fact? Second, the *transformation* aspect answers how a change to the input affects the truthfulness of the desired property.

There have been various approaches to support both these aspects in traditional database and datalog frameworks, some of which we explore next.

2.1 Database Repair and Integrity Constraints

One well-studied desired property in traditional database applications is **Consistency**. Arenas et al. (1999) define a database instance r as *consistent* if r satisfies a set of *integrity constraints* IC. They then logically characterize "consistent query answers in inconsistent databases". The intuitive interpretation here is that an answer to a query posed to a database that violates integrity constraints should be the same as that obtained from a minimally **repaired** version of the original database.

Example. Consider a student database. $Student(x, y, z)$ means that x is the student number, y is the student's name, and z is the address. The following ICs state that the first argument is the relation's key:

$$\forall(x, y, z, u, v)(Student(x, y, z) \wedge Student(x, u, v) \supset y = u),$$
$$\forall(x, y, z, u, v)(Student(x, y, z) \wedge Student(x, u, v) \supset z = v).$$

Student
S1 N1 D1
S1 N2 D1

Course
S1 C1 G1
S1 C2 G2

The inconsistent database instance r (shown) has two repairs, each removing one of the tuples in $Student$. Considering all repairs, for a query $\exists z Course(S1, y, z)$, we obtain $C1$ and $C2$ as the consistent answers. However, for $\exists(u, v(Student(u, N1, v) \wedge Course(u, x, y))$ we obtain no (consistent) answers.

This is interesting since a *repaired* database r' was transformed to satisfy the desired integrity constraints IC. Of course, there could be many repairs for an inconsistent database. Arenas et al. (1999) hence propose a solution to retrieve consistent answers that use only the original database (despite inconsistency). The idea is to syntactically transform a query Q, reinforcing residues of ICs locally, to a query Q'. Evaluating Q' on the original database returns the set of consistent answers to the query Q. This avoids explicitly computing the repairs. Here's an example:

Example. Consider the integrity constraint $\forall x(\neg P(x) \vee Q(x))$. If $Q(x)$ is false, then $\neg P(x)$ must be true. So, when querying $\neg Q(x)$, we make sure to generate the query $\neg Q(x) \wedge \boxed{\neg P(x)}$ where the highlighted part is the residue added.

$$\dfrac{\dfrac{\texttt{new}(a,l_1)}{\texttt{vpt}(a,l_1)} \qquad \dfrac{\texttt{assign}(b,a) \qquad \dfrac{\texttt{new}(a,l_1)}{\texttt{vpt}(a,l_1)}}{\texttt{vpt}(b,l_1)} \qquad a \neq b}{\texttt{alias}(a,b)}$$

Figure 2: Full proof tree for $\texttt{alias}(a,b)$ in an alias analysis.

Challenges: There are challenges in extending this directly to Datalog and program transformations. Mainly, queries and constraints here are limited to a fragment of first-order logic and rewritten into a new query in the same language. For instance, completeness is lost when it is applied to disjunctive or existential queries. More work (Arenas et al., 2003; Bertossi, 2011) along these lines have attempted to extend writing repair strategies in logical languages like Datalog, which are more expressive than FO logic. However, scalability issues exist when writing and applying these to large rulesets, as in static analysis.

A bigger challenge in (directly) connecting this to Datalog for program transformation is that these approaches circumvent computing repairs entirely. This is of little applicability when the goal is to find a transformation.

2.2 Data Provenance and Debugging Datalog

While connections to database repair are apparent, another way to view transformations modulo a desired property is to look into **Data Provenance** for debugging why a property holds. Despite the numerous advantages, the declarative semantics of Datalog poses a debugging challenge. Logic specifications lack the notion of state and state transitions. After evaluation, we can only view relations in their entirety without explaining the origin or derivation of data.

A standard solution for these explanations is a *proof tree* (Abiteboul et al., 1995). A proof tree for a tuple describes the derivation of that tuple from input tuples and rules. A valid proof tree (all nodes hold) can explain an unexpected tuple when debugging. A failed proof tree provides insight into why a tuple is not produced. Figure 2 shows an example of a proof tree for an alias analysis.

Challenges: However, these state-of-the-art techniques do not scale to large program analysis problems. Firstly, unlike top-down evaluation, scalable bottom-up Datalog evaluation does not have a notion of proof trees. Consequently, techniques propose rewriting the Datalog specification with provenance information (Deutch et al., 2015; Köhler et al., 2012; Lee et al., 2017). Here, a common issue is the need for re-evaluation when debugging, which can be expensive for industrial-scale static analysis problems (think something as precise as DOOP (Bravenboer & Smaragdakis, 2009), which may take multiple days for medium-to-large Java programs). Another major challenge is infinitely many proof trees and their conciseness–brute force search is infeasible, and storing proof trees during evaluation is memory intensive.

Recently, Zhao et al. (2020) proposed storing *proof annotations* alongside tuples that include the height of the minimal proof tree and the rule that generated the tuple. Further, they extend the standard subset lattice of bottom-up evaluation as follows:

1. Define a *provenance instance* (I, h) as an instance of tuples I along with a function h that provides a height annotation for each tuple in the instance.
2. Define a *provenance lattice* as one that follows the ordering:

$$(I_1, h_1) \sqsubseteq (I_2, h_2) \iff I_1 \subseteq I_2 \textit{ and } \forall t \in I_1 : h_1(t) \geq h_2(t).$$

3. Define the join of instances (I, h) and (I', h') as $(I \cup I', h'')$ where h'' is the minimum of h and h' for each tuple in the join.

Since bottom-up evaluation is equivalent to applying a monotone function to move "up" a lattice, this guarantees the minimality of these height annotations. During debugging, one can reconstruct one level of the minimal proof for a tuple t using a top-down search. While we omit details here, the example below illustrates it:

Example. Say $\texttt{alias}(a, b)$ is the tuple of interest. From bottom-up evaluation, the height annotation is $h(\texttt{alias}(a, b)) = 4$, and the generating rule is `alias(X,Y) :- vpt(X,o), vpt(Y,o)`. Now, we can search for tuples for the body of this rule such that $\texttt{X} = a$, $\texttt{Y} = b$, $h(\texttt{vpt}(a, o)) < 4$, and $h(\texttt{vpt}(b, o)) < 4$. Finding these tuples forms a one-level proof tree.

Zhao et al. (2020) find that this adds minimal overhead to Datalog evaluation while enabling debugging, even in Doop-like analysis generating millions of output tuples. This approach could take us one step further from program analysis to transformations with declarative approaches. Granularly reflecting on why we inferred a fact is particularly useful; it hints at potential transformation (repair) strategies. Notably, this has been integrated into state-of-the-art Datalog engines like Soufflé [1]. However, an aspect that remains unanswered is executing these transformations and viewing their effects.

2.3 Incrementality and Datalog

The incrementality of datalog engines allows for analysis on the fly, on-demand, and online analysis. Interactive applications using this benefit could provide insights into how to reason EDB transformations.

Example. As an example, program analysis frameworks like Doop (Bravenboer & Smaragdakis, 2009) evaluate a set of rules defined over the abstract syntax tree of the program. Integrating this into an IDE requires re-evaluating the rules after every few keystrokes (note, an ongoing transformation). The evaluation must preserve intermediate results efficiently to achieve interactive performance.

Datalog dialects such as Differential Datalog (DDLog) (Ryzhyk & Budiu, 2019) have been proposed to make writing such applications easier. A DDlog programmer writes traditional, non-incremental Datalog programs. However, DDlog's execution model is fully incremental: at runtime, DDlog programs receive streams of changes to the input relations and produce streams of corresponding changes to derived relations.

This capability significantly benefits tools like Language Server Protocols (LSP) servers and compilers, providing developers with real-time feedback and suggestions. However, this solves only part of the problem, where one can reason about the effect of changes. What remains is the part where we *reflect* on what change would provide the desired effect.

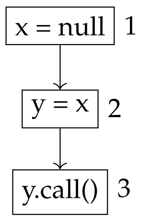
Elastic incrementalization (Zhao et al., 2021; 2023) extends this by incorporating provenance annotations for input adjustments in incremental Datalog. They switch between a low-overhead Bootstrap strategy that targets high-impact updates and an Update strategy that targets low-impact updates.

3 A Symbolic Execution Perspective

In the previous sections, we cover various viewpoints on the challenge of reflecting on input transformations modulo a desired property in a Datalog setting. Some solve the transformation aspect (Database repair), and others the reflection aspect (Data provenance, Debugging, and Incremental Datalog). However, a unification of the two within standard Datalog remains to bridge the use of Datalog from pure program analyses to transformations. Consequently, one must be able to represent changes to the input and then execute them to view how they affect the output, all in standard Datalog. Hence, the key missing pieces are:

1. How to represent a change $\Delta(EDB)$ in standard Datalog?
2. How to execute $\Delta(EDB)$ in standard Datalog, s.t., we can reflect on a map $\Delta(EDB) \mapsto \Delta(IDB)$?

[1] https://souffle-lang.github.io/provenance

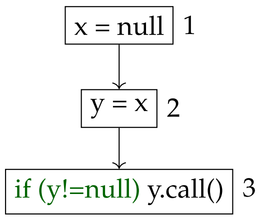


```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
call("y", 3).
```

(a) Program's CFG and EDB

```
npe(V,L):-
    call(V,L),
    null(V,L),
    !guard(V,L).
null(V,L):-
    flow(L1,L),
    assign_null(V,L1).
null(V,L):-
    flow(L1,L),
    null(V,L1),
    !assign(V,_,L1),
    !assign_obj(V,L1).
null(V,L):-
    flow(L1,L),
    assign(V,V1,L1),
    null(V1,L1).
```

(b) Datalog query for NPEs



```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
+ guard("y", 3)
call("y", 3).
```

(c) Repair and EDB change

Figure 3: A program with a null pointer exception (NPE), an analysis that detects it, and a repair that fixes it–all visualized in a Datalog setting for program analyses.

Liu et al. (2023) presented one solution. Liu et al. (2023) discuss an instantiation of the larger problem in program repair (a specific program transformation). Figure 3 illustrates an example of the setup. With this setup, one can identify bugs (such as null pointer exceptions) using a datalog query over program facts. The result is an observed fact, say `npe("y", 3)`, that suggests the presence of the bug. Additionally, as seen in Section 2.2 and Figure 2, one can derive a proof tree that describes the derivation of the observed fact.

3.1 Motivation: SymEx for Program Repair

Popular research in automated program repair uses symbolic execution and testing. Nguyen et al. (2013) proposed SEMFIX, where the authors reformulated the requirement on the repaired code to pass a given set of tests as a constraint. SEMFIX generates repair constraint via controlled symbolic execution (Boyer et al., 1975) of the program. Such a constraint is then solved by iterating over a space of repair expressions. Here, the key idea is in modeling the problem: abstract a given program by injecting symbols and executing the program symbolically to infer repair constraints. Solving these constraints results in potential repairs.

Liu et al. (2023) extend this idea to Datalog. They introduce the notion of **Symbolic Execution of Datalog** (SEDL) that determines how a change to the EDB affects the output of a given query, the details of which are described next.

3.2 Symbolic Execution of Datalog

A set of changes to the database can be encoded using symbols. Liu et al. (2023) introduce three types of symbols: (1) symbolic constants (α's) that represent unknown constants, (2) symbolic predicates (*rho*'s) for unknown predicates, and (3) symbolic signs (ξ's) for unknown truthfulness of facts. These symbols can be used to inject and encode a large space of changes to the EDB, as shown below:

Example. For instance, consider the EDB in Figure 3. Injecting symbolic facts into it could result in a symbolic EDB as follows:

ξ_1 `flow(1, 2).`
ξ_2 `flow(2, 3).`
ξ_3 flow(α_1, α_2).
ξ_4 flow(α_2, α_3).
`assign_null("x", 1).`
`assign_null("x", 1).`
`call("y", 3).`
ξ_5 flow(α_4,, 3).
ξ_6 assign_obj(α_5, α_6).

Any valuation of these symbols corresponds to a concrete EDB (including the original one). Next, these sets of changes (encoded as symbols) are "executed" using standard Datalog. To do so, Liu et al. (2023) propose a meta-programming approach. They encode the query and symbolic EDB into a meta-program using transformation rules. The meta-program uses auxiliary variables in the relations to capture the bindings for the symbols introduced. An example is discussed below:

Example. Consider the rule `null(V,L):- flow(L1, L), assign_null(V, L1).` that calculates when a variable is `null`. Each predicate in a rule like this is augmented with variables for each symbol introduced. These auxiliary variables store the assignments to their corresponding symbols:

```
null(V,L, C1, ..., Cn):- flow(L1, L, C1, ..., Cn), assign_null(V, L1, C1, ..., Cn);
```

As shown, the rules also propagate the values bound to these symbols. Similarly, a symbolic EDB fact such as `flow(`α_1`, 2)` is converted to a rule that enumerates values from the domain of the symbol used:

`flow(C1, 2, C1, ..., Cn) :- dom_`α_1`(C1), ..., dom_`α_n`(Cn)`

When executed, the values in these binding variables will capture the **assignment constraints that should hold** for the corresponding fact. A similar strategy is used to bind symbolic predicates that identify **which predicate instantiation is needed** to infer a fact. Lastly, bindings for symbolic signs represent whether the output fact **can be inferred with or without relying on a symbolic signed fact**.

The overall semantics of symbolically executing a datalog program P can thus be described as a function $\mathcal{S}_P : 2^{SEHB} \to 2^{SIHB \times \Phi}$. That is, each inferred fact in the symbolic intentional database (SIHB) is accompanied by an **Inference Condition** Φ that summarizes the values of auxiliary variables. This condition defines under what constraint the output fact generated (similar to path conditions in conventional symbolic execution). For instance, the fact $npe(y, 3)$ could be accompanied by the following condition:

over nodes in proof tree   from aux vars for each node

$$\text{npe(y, 3);} \quad \Phi \quad = \quad \begin{array}{l} \xi_1 \wedge \xi_2 \wedge \neg(\xi_5 \wedge \alpha_4 = \text{“y”}) \\ \vee(\xi_2 \wedge \xi_3 \wedge \alpha_1 = 1 \wedge \xi_4 \wedge \alpha_3 = 2) \\ \quad\quad \wedge\neg(\xi_6 \wedge \alpha_5 = \text{“x”} \wedge \alpha_6 = \alpha_2) \\ \quad\quad \wedge\neg(\xi_5 \wedge \alpha_4 = \text{“y”}) \\ \vee \cdots \end{array}$$

over a subset of programs w/ same proof

Consequently, $\neg\Phi$ represents the constraint under which the corresponding fact is not true. This closes the loop for program repair, since $\neg\Phi$ is essentially a **repair constraint**–any solution for it is a valid repair or change to the EDB. These perspectives and ideas introduced to Datalog could potentially be extended beyond program repair.

4 Conclusion

In this paper, we survey existing work to answer the question: *Can program transformations be translated to useful aspects of Datalog?* In light of this question, we looked at connections to databases in two directions. First, we studied the origins or derivation of a fact, found existing connections to database repair modulo integrity constraints, and unveiled challenges with them. Further, we find research on data provenance, Datalog augmented with lattices, and debugging Datalog quite relevant to the question at hand. Finally, we cover recent efforts in program repair (an example of a program transformation) that propose a symbolic execution perspective to this problem. Overall, this survey describes and connects several related ideas and provides a basis for future research on Datalog-based program transformations.

References

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 68–79, 1999.

Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory and practice of logic programming*, 3(4-5): 393–424, 2003.

Leopoldo Bertossi. *Database Repairs and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 243–262, 2009.

Daniel Deutch, Amir Gilad, and Yuval Moskovitch. Selective provenance for datalog programs using top-k queries. *Proceedings of the VLDB Endowment*, 8(12):1394–1405, 2015.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pp. 422–430. Springer, 2016.

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. Declarative datalog debugging for mere mortals. In *International Datalog 2.0 Workshop*, pp. 111–122. Springer, 2012.

Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1–12, 2005.

Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. Efficiently computing provenance graphs for queries with negation. *arXiv preprint arXiv:1701.05699*, 2017.

Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. Program repair guided by datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1216–1228, 2023.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781. IEEE, 2013.

Thomas W Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, pp. 163–196. Springer, 1995.

Leonid Ryzhyk and Mihai Budiu. Differential datalog. *Datalog*, 2:4–5, 2019.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4): 58–66, 2018.

John Whaley and Monica S Lam. 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*, pp. 131–144, 2004.

David Zhao, Pavle Subotić, and Bernhard Scholz. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(2):1–35, 2020.

David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. Towards elastic incrementalization for datalog. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, pp. 1–16, 2021.

David Zhao, Pavle Subotić, Mukund Raghothaman, and Bernhard Scholz. Automatic rollback suggestions for incremental datalog evaluation. In *International Symposium on Practical Aspects of Declarative Languages*, pp. 295–312. Springer, 2023.