Declarative Datalog Debugging for Mere Mortals

Sven Köhler[1], Bertram Ludäscher[1], and Yannis Smaragdakis[2]

[1] Department of Computer Science, University of California, Davis
{svkoehler,ludaesch}@ucdavis.edu
[2] LogicBlox, Inc., Atlanta, GA and Univ. of Athens, Greece
yannis.smaragdakis@logicblox.com

Abstract. Tracing why a "faulty" fact A is in the model $M = P(I)$ of program P on input I quickly gets tedious, even for small examples. We propose a simple method for debugging and "logically profiling" P by generating a provenance-enriched rewriting $\hat{P}$, which records rule firings according to the logical semantics. The resulting provenance graph can be easily queried and analyzed using a set of predefined and ad-hoc queries. We have prototypically implemented our approach for two different Datalog engines (DLV and LogicBlox), demonstrating the simplicity, effectiveness, and system-independent nature of our method.

1 Introduction

Developing declarative, rule-based programs can be surprisingly difficult in practice, despite (or because of) their declarative semantics. Possible reasons include what Kunen long-ago called *the PhD effect* [10], i.e., that a PhD in logic seems necessary to understand the meaning of certain logic programs (with negation). Similarly, *An Amateur's Introduction to Recursive Query Processing* [2] from the early days of deductive databases, rather seems to be for experts only. So what is the situation now, decades later, for a brave, aspiring Datalog 2.0 programmer who wants to develop complex programs?

The meaning and termination behavior of a *Prolog* program P depends on, among other things, the order of rules in P, the order of subgoals within rules, and even (apparently minor) updates to base facts. Consider, e.g., the program for computing the transitive closure of a directed graph, i.e., $P_{\mathsf{tc}} =$

$$\begin{aligned} r_1\colon&\quad \mathsf{tc}(X,Y) \text{ :- } \mathsf{e}(X,Y).\\ r_2\colon&\quad \mathsf{tc}(X,Z) \text{ :- } \mathsf{e}(X,Y), \mathsf{tc}(Y,Z).\end{aligned}$$

Seasoned logic programmers know that P_{tc} is *not* a correct way to compute the transitive closure in Prolog.[3] Under a more declarative *Datalog* semantics, on the other hand, P_{tc} indeed *is* correct, since the result does *not* depend on rule or subgoal order. The flip side, however, is that effective and practically useful

[3] For $I = \{\mathsf{e}(\mathsf{a},\mathsf{b}), \mathsf{e}(\mathsf{b},\mathsf{a})\}$ the query ?-tc(c,X) correctly returns "No", while the similar ?-tc(X,c) will not terminate! Prolog's behavior gets worse when swapping rules r_1 and r_2, or when using left- or doubly-recursive variants P^l_{tc}, P^d_{tc}, respectively.

procedural debugging techniques for Prolog, based on the *box model* [19], are not available in Datalog. Instead, new debugging techniques are needed that are solely based on the declarative reading of rules. In this paper, we develop such a framework for declarative debugging and logic profiling.

Let $M{=}P(I)$ be the model of P on input I. Bugs in P (or I) manifest themselves through unexpected answers (ground atoms) $A \in M$, or expected but missing $A \notin M$. The key idea of our approach is to rewrite P into a *provenance-enriched* program $\hat{P}$, which records the derivation history of $M{=}P(I)$ in an extended model $\hat{M}{=}\hat{P}(I)$. A provenance graph G is then extracted from $\hat{M}$, which the user can explore further via predefined views and ad-hoc queries.

Use Cases Overview. Given an IDB atom A, our approach allows to answer questions such as the following: What is the *data lineage* of A, i.e., the set of EDB facts that were used in a derivation of A, and what is the *rule lineage*, i.e., the set of rules used to derive A? When chasing a bug or trying to locate a source of inefficiency, a user can explore further details: What is the graph structure G_A of all derivations of A? What is the *length* of A, i.e., of shortest derivations, and what is the *weight*, i.e., number of simple derivations (proof trees) of A?

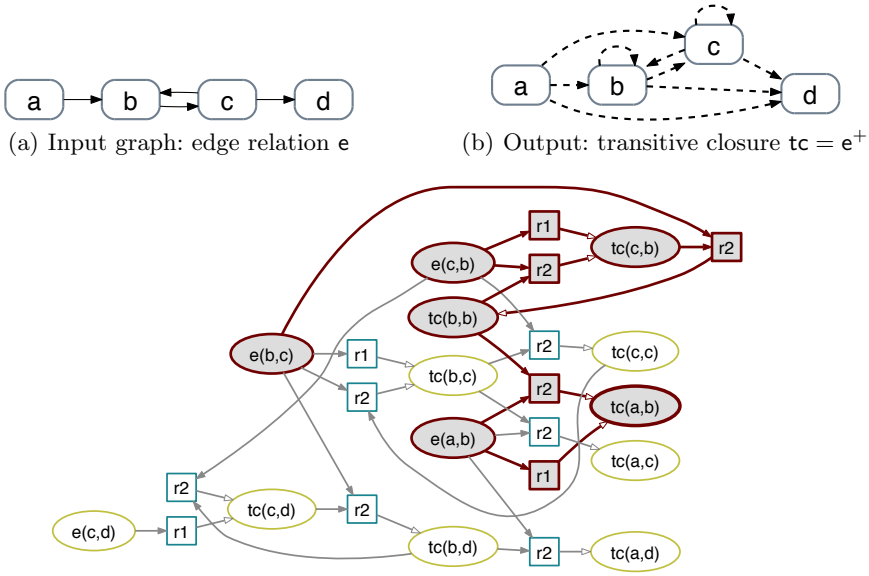
For another example, assume the user encounters two "suspicous" atoms A and B. It is easy to compute the *common lineage* $G_{AB} = G_A \cap G_B$ shared by A and B, or the *lowest common ancestors* of A and B, i.e., the rule firings and ground atoms that occur "closest" to A and B in G_{AB}, thus triangulating possible sources of error, somewhat similar to ideas used in delta debugging [22].

Since nodes in G_A are associated with relation symbols and rules, a user might also want to compute other aggregates, i.e., not only at the level of G_A (ground atoms and firings), but at the level of (non-ground) rules and relation symbols, respectively. Through this *schema-level profiling*, a user can quickly find the hot (cold) spots in P, e.g., rules having the most (least) number of firings.

Running Example. Figure 1 gives an overview using a very simple example: (a) depicts an input graph e, while (b) shows its transitive closure $\mathsf{tc} := \mathsf{e}^+$. The structure and number of distinct derivations of tc atoms from base edges in e can be very different, e.g., when comparing the right-recursive P_{tc} $(=P^r_{\mathsf{tc}})$ above, with left-recursive or doubly-recursive variants P^l_{tc} or P^d_{tc}, respectively.

The provenance graph G (or the relevant subgraph $G_A \subseteq G$, given a goal A) provides crucial information to answer the above use cases. Fig. 1(c) shows the provenance graph for the computation of tc via P^r_{tc} from above. Box nodes represent *rule firings*, i.e., individual applications of the *immediate consequence operator* T_P, and connect all body atoms to the head atom via a unique firing node. For the debug goal $A = \mathsf{tc}(\mathsf{a}, \mathsf{b})$ the subgraph G_A, capturing all possible derivations of A, is highlighted (through filled nodes and bold edges).

Overview and Contributions. We present a simple method for debugging and logically profiling a Datalog program P via a provenance-enriched rewriting $\hat{P}$. The key idea is to extract from the extended model $\hat{M}$ a provenance graph G which is then queried, analyzed, and visualized by the user. Given a debug goal A, relevant subgraphs G_A can be obtained easily and further analyzed via a library of common debug views and ad-hoc user queries. At the core of our approach



(a) Input graph: edge relation e

(b) Output: transitive closure $\mathsf{tc} = \mathsf{e}^+$

(c) Provenance graph G; highlighted subgraph $G_{\mathsf{tc(a,b)}}$; firing nodes r (boxes), atom nodes A (ovals); edge types $A \xrightarrow{\text{in}} r$ and $r \xrightarrow{\text{out}} A$ (edge labels not shown)

Fig. 1. P^r_{tc}-provenance graph for input e, with derivations of tc(a,b) highlighted in (c).

are rewritings that (i) capture rule firings, then (ii) reify them, i.e., turn them into nodes in G (via Skolem functions), while (iii) keeping track of derivation lengths using Statelog [11], a Datalog variant with states.[4] The rewritten Statelog program $\hat{P}$ is state-stratified [13] and has PTIME *data complexity*. We view the simplicity and system-independence as an important benefit of our approach. We have rapidly prototyped this approach for rather different Datalog engines, i.e., DLV [12] and LogicBlox [14], and are currently developing improved versions. We also note a close relationship of our provenance graphs with *provenance semirings* [8] (a detailed account is beyond the scope of this paper). Here our focus is on presenting a simple, effective method for debugging and profiling declarative rules for "mere mortals".

2 Provenance Rewritings for Datalog

In this section, we present three Datalog rewritings $P \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \hat{P}$ for capturing rule firings, graph generation, and Statelog evaluation, respectively. We assume the reader is familiar with Datalog (e.g., see [1, 17]); the resulting Statelog program $\hat{P}$ has PTIME data complexity and involves a limited (i.e., safe) form of Skolem functions and state terms [13]. In the sequel, $\bar{X} = X_1, \ldots, X_n$ denotes a variable vector; lower case terms $a, b, \ldots$ denote constants.

[4] Other state-oriented Datalog extensions include Datalog$_{nS}$ [6] and XY-Datalog [21].

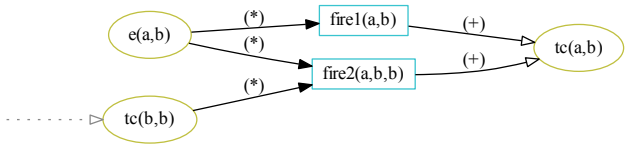


Fig. 2. Subgraph with two rule firings $\mathsf{fire}_1(\mathsf{a},\mathsf{b})$ and $\mathsf{fire}_2(\mathsf{a},\mathsf{b},\mathsf{b})$, both deriving $\mathsf{tc(a,b)}$.

2.1 Recording Rule Firings: $P \overset{F}{\rightsquigarrow} P^F$

The first rewriting (cf. Green et al. [9]), captures the provenance of rule firings. Let r be a unique identifier of a rule in P. We assume r to be *safe*, i.e., every variable in r must also occur positively in the body:

$$r: \quad H(\bar{Y}) \text{ :- } B_1(\bar{X}_1), \ldots, B_n(\bar{X}_n)$$

Let $\bar{X} := \bigcup_i \bar{X}_i$ include all variables in r, ordered, e.g., by occurrence in the body. Since r is safe, $\bar{Y} \subseteq \bar{X}$, i.e., the head variables are among the $\bar{X}$. The rule r is now replaced by two new rules in the rewritten program P^F:

$$\begin{aligned} r_{in} &: \mathsf{fire}_r(\bar{X}) \text{ :- } B_1(\bar{X}_1), \ldots, B_n(\bar{X}_n) \\ r_{out} &: \quad H(\bar{Y}) \text{ :- } \mathsf{fire}_r(\bar{X}) \end{aligned}$$

Thus P^F records, for each r-satisfying instance $\bar{x}$ of $\bar{X}$, a unique fact: $\mathsf{fire}_r(\bar{x})$.

2.2 Graph Reification of Firings: $P^F \overset{G}{\rightsquigarrow} P^G$

To facilitate querying the results of the previous step, we *reify* ground atoms and firings as nodes in a *labeled provenance graph* G. For each pair of rules r_{in}, r_{out} above, we add n rules ($i = 1, \ldots, n$) to generate the in-labeled edges in G:

$$\mathsf{g}(\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \text{ :- } \mathsf{fire}_r(\bar{X})$$

and one more rule for generating out-labeled edges in G as well:

$$\mathsf{g}(\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \text{ :- } \mathsf{fire}_r(\bar{X})$$

Note the safe use of atoms as *Skolem terms* in the rule heads: for finitely many rule firings $\mathsf{fire}_r(\bar{x})$, we obtain a finite number of in- and out-edges in G.

Example. After applying both transformations $\cdot \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot$ to P_{tc} from above, the rewritten program P_{tc}^G can be executed, yielding a directed graph with labeled edges $\mathsf{g}(v_1, \ell, v_2)$ in the enriched model $\hat{M}$. Figure 2 shows a subgraph with two rule firings, both deriving the atom $\mathsf{tc(a,b)}$. Oval (yellow) nodes represent atoms A and boxed (blue) nodes represent firings F. Arrows with solid heads and label $(*)$ are in-edges, while those with empty heads and label $(+)$, represent out-edges. Note that according to the declarative semantics, in (out) edges, model logical conjunction "$\wedge$" (logical disjunction "$\vee$"), respectively. Thus, w.r.t. their incoming edges, boxed nodes are AND-nodes, while oval nodes are OR-nodes.[5]

[5] In semiring parlance, they are product "$\otimes$" and sum "$\oplus$" nodes, respectively.

2.3 Statelog Rewriting: $P^G \overset{S}{\rightsquigarrow} P^S$

Statelog [11, 13] is a state-oriented Datalog variant for expressing active update rules and declarative rules in a unified framework. The next rewriting simulates a Statelog derivation in Datalog via a limited (safe) form of "state-generation". The key idea is to keep track of the firing rounds $I_{n+1} := T_P(I_n)$ of the T_P operator ($I_0 := I$ is the input database). This provides a simple yet powerful means to detect tuple rederivations, to identify unfounded derivations, etc.

First, replace all rules r_{in}, r_{out} above with their state-oriented counterparts:

$$r_{in} : \mathsf{fire}_r(\mathsf{S1}, \bar{X}) \text{ :- } B_1(\mathsf{S}, \bar{X}_1), \ldots, B_n(\mathsf{S}, \bar{X}_n),\ \mathsf{next}(\mathsf{S}, \mathsf{S1}).$$
$$r_{out} : \quad H(\mathsf{S}, \bar{Y}) \text{ :- } \mathsf{fire}_r(\mathsf{S}, \bar{X}).$$

The goal $\mathsf{next}(\mathsf{S}, \mathsf{S1})$ is used for the safe generation of new states: The next state $s{+}1$ is generated only if at least one atom A was new in state s :

$$\mathsf{next}(0, 1) \text{ :- } \mathsf{true}.$$
$$\mathsf{next}(\mathsf{S}, \mathsf{S1}) \text{ :- } \mathsf{next}(\_, \mathsf{S}),\ \mathsf{new}(\mathsf{S}, A),\ \mathsf{S1} := \mathsf{S} + 1.$$

An atom A is newly derived if it is true in $s{+}1$, but not in the previous state s:

$$\mathsf{newAtom}(\mathsf{S1}, A) \text{ :- } \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, \_, \mathsf{out}, A),\ \neg\, \mathsf{g}(\mathsf{S}, \_, \mathsf{out}, A).$$

Similarly, rule firing F is new if it is true in $s{+}1$, but not previously in s:

$$\mathsf{newFiring}(\mathsf{S1}, F) \text{ :- } \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, F, \mathsf{out}, \_),\ \neg\, \mathsf{g}(\mathsf{S}, F, \mathsf{out}, \_).$$

The n rules for generating in-edges are replaced with state-oriented versions:

$$\mathsf{g}(\ \mathsf{S},\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \text{ :- } \mathsf{fire}_r(\mathsf{S},\ \bar{X})$$

and similarly, for the out-edge generating rules:

$$\mathsf{g}(\ \mathsf{S},\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \text{ :- } \mathsf{fire}_r(\mathsf{S},\ \bar{X}).$$

It is not difficult to see that the above rules are state-stratified (a form of local stratification) and that the resulting program terminates after polynomially many steps [13]: When no more new atoms (or firings) are derived in a state, then the above rules for next can no longer generate new states, thus in turn preventing rules of type r_{in} from generating new $\mathsf{fire}_r(\mathsf{S1}, \bar{X})$ atoms.

Example. When applying the transformations $\cdot \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \cdot$ to the transitive closure program P^r_{tc}, a 4-ary graph g is created, with the additional T_p-round counter in the first (state) argument position. Figure 3 shows the graphical representation of g for our running example (observe the cycle in g, caused by the cycle in the input e).

3 Debugging and Profiling using Provenance Graphs

When debugging and profiling Datalog programs we typically employ all program transformations $P \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \hat{P}$, i.e., the enriched model $\hat{M}$ contains the full provenance graph relation g with state annotations.

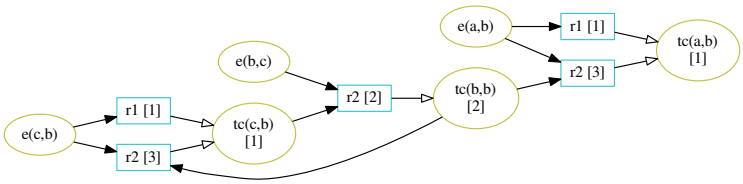


Fig. 3. State-annotated provenance graph g for the derivation of tc(a,b). Annotations [in brackets] show the firing round (i.e., state number) in which an atom was first derived. To avoid clutter, we often depict firing nodes without their variable bindings.

3.1 Debugging Declarative Rules

If a Datalog program does not compute the expected model, it is very helpful to understand how the model was derived, focusing in particular on certain goal atoms during the debugging process. Since relation g captures all possible derivations of the given program, we can define various queries and views on g to support debugging.

Provenance Graph. The complete description of how a program was evaluated can be derived by just visualizing the whole provenance graph g, optionally removing the state argument through projection:

```
ProvGraph(X,L,Y) :- g(_,X,L,Y).
```

Figure 1(c) shows the provenance graph for a transitive closure computation. One can easily see how all transitive edges were derived and—by following the edges backwards—one can see on which EDB facts and rules each edge depends.

Provenance Views. Since provenance graphs are large in practice, it is often desirable to just visualize subgraphs of interest. The following debug view returns all "upstream" edges, i.e., the provenance subgraph relevant for debug atom Q:

```
ProvView(Q,X,out,Q) :- ProvGraph(X,out,Q).
ProvView(Q,X,L,Y) :- ProvView(Q,Y,_,_), ProvGraph(X,L,Y).
```

Figure 3 shows the result of this view for the debug goal $Q = \mathsf{tc}(\mathsf{a},\mathsf{b})$; the large (goal-irrelevant) remainder of the graph is excluded. Fig. 1(c), in contrast, shows the same query but now in the context of the whole provenance graph.

Computing the Length of Derivations. A typical question during debugging is when and from which other facts a debug goal was derived. Such temporal questions can be explained using a Statelog rewriting of the program. We annotate atoms and firings with a *length* attribute to record in which round they were first derived. The length is defined as follows:

$$\mathsf{len}(F) := 1 + \max\{\ \mathsf{len}(A) \mid (A \xrightarrow{\mathsf{in}} F) \in \mathsf{g}\ \}\ ; \quad \text{if } F \text{ is a firing node}$$

$$\mathsf{len}(A) := \begin{cases} \min\{\ \mathsf{len}(F) \mid (F \xrightarrow{\mathsf{out}} A) \in \mathsf{g}\ \} & ; \quad \text{if } A \text{ is an IDB atom} \\ 0 & ; \quad \text{if } A \text{ is an EDB atom} \end{cases}$$

A rule firing F can only succeed one round after the *last* body atom (i.e., having maximal length) has been derived. Conversely, the length of an atom A is determined by its *first* derivation (i.e., having minimal length). The Statelog rewriting captures evaluation rounds, so the state associated with a *new* firing determines the length of the firing:

```
len(F,LenF) :− newFiring(S,F), LenF=S.
```

Similarly, the length of an atom is equal to the first round it was derived:

```
len(A,LenA) :− newAtom(S,A), LenA=S.
```

Fig. 4 shows a provenance graph with such length annotations.

3.2 Logic-Based Profiling

There are multiple ways to write a Datalog program that computes a desired query result and the performance of these programs may vary significantly. For example, consider an EDB with a linear graph e having 10 nodes. In addition to the right-recursive program P^r_{tc}, we consider the doubly-recursive variant P^d_{tc} with the rules: (1) $\mathsf{tc}(X,Y) :- \mathsf{e}(X,Y)$ and (2) $\mathsf{tc}(X,Y) :- \mathsf{tc}(X,Z), \mathsf{tc}(Z,Y)$. When computing tc, the two programs perform differently and we would like to find the cause. In this section, we present queries for profiling measures of Datalog programs that can help to answer such questions.

Counting Facts. When evaluating a Datalog program on an input EDB, a number of IDB atoms are derived. We assume here that the resulting model only contains desired facts, i.e., the program was already debugged with the methods described earlier. We can use, e.g., the number of derived IDB atoms as a baseline for profiling a program. It can be computed easily via aggregation:

```
DerivedFact(H) :− ProvGraph(_,out,H).
DerivedHeadCount(C) :− C = count{ H : DerivedFact(H) }.
```

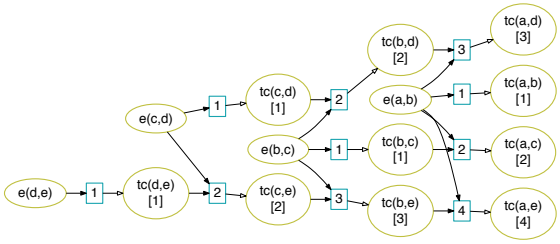
Both P^r_{tc} and P^d_{tc} derive 45 facts for our small graph example, which is exactly the number of transitive edges we are looking for.

Counting Firings. An important measure in declarative profiling is the number of rule firings needed to produce the final model. It can be computed from the out-edges and another simple aggregation:

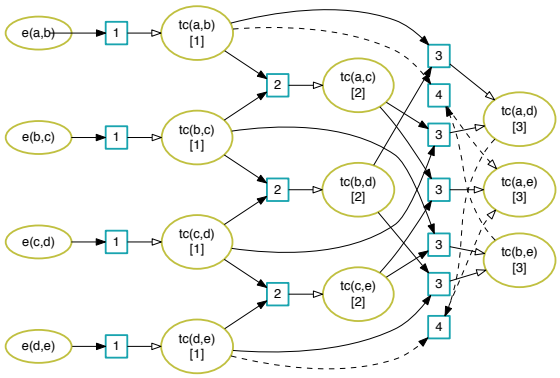
```
Firing(F) :− ProvGraph(F,out,_).
FiringCount(C) :− C = count{ F : Firing(F) }.
```

Using this measure we can see a clear difference between the two variants of the transitive closure program. While the right-recursive program P^r_{tc} uses 45 rule firings to compute the model, the doubly-recursive variant P^d_{tc} causes 129 rule firings to derive the same 45 transitive edges. The reason is that P^d_{tc} will use all combinations of edges to derive a fact, while P^r_{tc} extends paths only in one direction, one edge at a time.

For better readability, Figure 4 shows the provenance graph for a smaller input graph consisting of a 5-node linear chain. Nodes are annotated with their



(a) P^r_{tc}: right-recursive



(b) P^d_{tc}: doubly-recursive

Fig. 4. Provenance graphs with annotations for profiling P^r_{tc} and P^d_{tc} on a 5-node linear graph. P^d_{tc} causes more rule firings than P^r_{tc} and also derives facts in multiple ways. Numbers denote $\mathsf{len}(F)$ (in firing nodes) and $\mathsf{len}(A)$ (in atom nodes), respectively.

length, i.e., earliest possible derivation round. Note how some atom nodes in the graph for P^d_{tc} in Fig. 4(b) have more incoming edges (and derivations) than the corresponding nodes in the P^r_{tc} variant shown in Fig. 4(a).

Computing the Maximum Round. Another measure is the number of states (T_P rounds), needed to derive all conclusions. From the rewriting P^S, we can simply determine the final state:

$$\mathsf{MaxRound(MR)} :- \mathsf{MR} = \mathsf{max}\{\ \mathsf{S} : \mathsf{g(S,\_,\_,\_)}\ \}.$$

This measure shows another clear difference between P^r_{tc} and P^d_{tc}: While P^r_{tc} requires 10 rounds to compute all transitive edges in our sample graph, P^d_{tc} only needs 6 rounds. Generally, the doubly-recursive variant requires significantly fewer rounds, i.e., logarithmic in the size of the longest simple path in the graph versus linear for the right-recursive implementation.

Counting Rederivations. To analyze the number of derivations in more detail, we can use the Statelog rewriting P^S to also capture temporals aspects. With each application of the T_P operator, some facts might be rederived. Note that, if a fact is derived via different variable bindings in the body of a rule (or different

rules), the rederivation is captured already in firings. Rederivations occurring during the fixpoint computation can be captured using the Statelog rewriting:

```
ReDerivation(S,F) :− g(S,F,out,A), len(A,LenA), LenA < S.
ReDerivationCount(S,C) :− C = count{ F : ReDerivation(S,F) }.
ReDerivationTotal(T) :− T = sum{ C : ReDerivationCount(S,C) }.
```

When comparing the rederivation counts, the difference between the P_{tc} variants is illuminated further. P^r_{tc} rederives facts 285 times until the fixpoint is reached. The double-recursive program P^d_{tc} causes 325 rederivations.

Schema-Level Profiling. We can easily determine how many new facts per *relation* are used in each round to derive new facts:

```
FactsInRound(S,R,A) :− g(S,A,in,_), RelationName(A,R).
FactsInRound(S1,R,A) :− g(S,_,out,A), next(S,S1), RelationName(A,R).
NewFacts(S,R,A) :− g(S,_,out,A), ¬ FactsInRound(S,R,A), RelationName(A,R).
NewFactsCount(S,R,C) :− C = count{ A : NewFacts(S,R,A) }.
```

This allows us to "plot" the temporal evolution for our result relation $\mathsf{R}(=\mathsf{tc})$: For P^r_{tc} and $\mathsf{S} = 1, \ldots, 9$ the counts are $\mathsf{C} = 9, 8, 7, 6, 5, 4, 3, 2, 1$, while for P^r_{tc} and $\mathsf{S} = 1, \ldots, 5$ we have $\mathsf{C} = 9, 8, 13, 14, 1$. Although P^d_{tc} requires fewer rounds than P^r_{tc}, its new fact count mostly increases over time, while for P^r_{tc}, it decreases.

Confronting the Real-World. In practical implementations, the doubly-recursive version P^d_{tc} has horrible performance. For a representative, realistic graph[6] with 1710 nodes and 3936 edges, the right-recursive P^r_{tc} runs in 2.6 sec, while the doubly-recursive P^d_{tc} takes 15.4 sec. Our metrics can easily explain the discrepancy. The tc-fact count for both versions is 304,000, but the rule firing count varies widely. Using our program rewriting and the profiling view FiringCount(C), we discover that P^d_{tc} has over 64 million different rule firings, while P^r_{tc} has under 566 thousand. One reason is that the doubly-recursive rule $\mathsf{tc}(X,Y) :- \mathsf{tc}(X,Z), \mathsf{tc}(Z,Y)$ derives the same $\mathsf{tc}(X,Y)$ fact many times over. This practical example also illustrates the burden of declarative debugging: Adding the profiling calculation to the right-recursive version, P^r_{tc} only grows the running time slightly, to 3 sec. Adding it to the doubly-recursive P^d_{tc} however, yields a running time of 51.3 sec. This is due to the cost of storing more than 64 million combinations of variables for rule firings.

4 GPAD Prototype Implementations

By design, our method of provenance-based debugging and profiling only relies on the declarative reading of rules, i.e., is agnostic about implementation details or

[6] The specifics are secondary to our argument, but we list them for completeness. The graph is the application-level call-graph and edges indicating whether a method can call another) for the `pmd` program from the DaCapo benchmark suite, as produced by a precise low-level program analysis. Timings are on a quad-core Xeon E5530 2.4GHz 64-bit machine (only one thread was active at a time) with plentiful RAM (24GB) for the analysis, using LogicBlox Datalog ver. 3.7.10.

evaluation techniques specific to the underlying Datalog engine. Indeed, parallel with the development of the method, we have implemented two incarnations of a ***G**raph-based **P**rovenance **A**nalyzer and **D**ebugger*, i.e., prototypes GPAD/DLV and GPAD/LB, for declarative debugging with the DLV [12] and LogicBlox [14] engines, respectively. Both prototypes "wrap" the underlying Datalog engine, and outsource some processing aspects to a host language.

For example, GPAD/DLV uses SWI-PROLOG [20] as a "glue" to automate (1) rule rewritings, (2) invocation of DLV, followed by (3) result post-processing, and (4) result visualization using Graphviz. We are actively developing GPAD further and plan a public release in the near future.

5 Related Work

Work on declarative debugging, in particular in the form of *algorithmic debugging* goes as far back as the 1980's [18, 7]. Algorithmic debugging is an interactive process where the user is asked to differentiate between the actual model of the (presumeably buggy) program and the user's intended model. Based on the user's input, the system then tries to locate the faulty rules in an interactive session. Our approach differs in a number of aspects. First, algorithmic debugging is usually based on a specific operational semantics, i.e., SLDNF resolution, a top-down, left-to-right strategy with backtracking and negation-as-failure, which differs significantly from the declarative Datalog semantics (cf. Section 1). Moreover, while our approach is applicable, in principle, in an interactive way, this suggests a tighter coupling between the debugger and the underlying rule engine. In contrast, our approach and its GPAD implementations do not require such tight coupling, but instead treat the rule engine as a black box. In this way, debugging becomes a post-mortem analysis of the provenance-enriched model $\hat{M} = \hat{P}(I)$ via simple yet powerful graph queries and aggregations.

Another approach, more closely related to ours, is the Datalog debugger [3], developed for the DES system. Unlike prior work, and similar to ours, they do not view derivations as SLD proof trees, but rather use a *computation graph*, similar to our labeled provenance graph. Our approach differs in a number of ways, e.g., our reification of derivations in a labeled graph allows us to use regular path queries to navigate the provenance graph, locate (least) common ancestors of buggy atoms, etc. Another difference is our use of Statelog for keeping track of derivation rounds, which facilitates profiling of the model computation over time (per firing round, identify the rules fired, the number of (re-)derivations per atom or relation, etc.) Recent related work also includes work on trace visualization for ASP [4], step-by-step execution of ASP programs [15], and an integrated debugging environment for DLV [16].

Debugging and Provenance. Chiticariu et al. [5] present a tool for debugging database schema mappings. They focus on the computation of derivation routes from source facts to a target. The method includes the computation of minimal routes, similar to shortest derivations in our graph. However, their approach

seems less conducive to profiling since, e.g., provenance information on firing rounds is not available in their approach.

There is an intruigingly close relationship between *provenance semirings*, i.e., provenance polynomials and formal power series [8], and our labeled provenance graphs G. The semiring provenance of atom A is represented in the structure of G_A. Consider, e.g., Figure 2: the in-edges of rule firings correspond to a logical conjunction "$\wedge$", or more abstractly, the product operator "$\otimes$" of the semiring. Similarly, out-edges represent a disjunction "$\vee$", i.e., an abstract sum operator "$\oplus$", mirroring the fact that atoms in general have multiple derivations. It is easy to see that a fact A has an infinite number of derivations (proof trees) iff there is a cycle in G_A: e.g., the derivation of $A = \mathsf{tc(a,b)}$ in Figures 1 and 2 involves a cycle through tc(b,b), tc(c,b), via two firings of r_2. This also explains Prolog's non-termination (Section 1), which "nicely" mirrors the fact that there are infinitely many proof trees. On the other hand, such cycles are not problematic in the original Datalog evaluation of $M = P(I)$ or in our extended provenance model $\hat{M} = \hat{P}(I)$, both of which can be shown to converge in polynomial time.

6 Conclusions

We have presented a framework for declarative debugging and profiling of Datalog programs. The key idea is to rewrite a program P into $\hat{P}$, which records the derivation history of $M = P(I)$ in an extended model $\hat{M} = \hat{P}(I)$. $\hat{P}$ is obtained from three simple rewritings for (1) recording rule firings, (2) reifying those into a labeled graph, while (3) keeping track of derivation rounds in the style of Statelog. After the rewritten program is evaluated, the resulting provenance graph can be queried and visualized for debugging and profiling purposes.

We have illustrated the declarative profiling approach by analyzing different, logically equivalent versions of the transitive closure program P_{tc}. The measures obtained through logic profiling correlate with runtime measures for a large, real-world example. Two prototypical systems GPAD/DLV and GPAD/LB have been implemented, for DLV and LogicBlox, respectively; a public release is planned for the near future. While we have presented our approach for positive Datalog only, it is not difficult to see how it can be extended, e.g., for well-founded Datalog. Indeed, the GPAD prototypes already support the handling of well-founded negation through a simple Statelog encoding [11, 13].

Acknowledgments. Work supported in part by NSF awards OCI-0722079, IIS-1118088, and a gift from LogicBlox, Inc.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[2] Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies. In: Readings in Database Systems. pp. 507–555 (1988)

[3] Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A theoretical framework for the declarative debugging of Datalog programs. Semantics in Data and Knowledge Bases pp. 143–159 (2008)
[4] Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Workshop on Software Engineering for Answer Set Programming (SEA) (2009)
[5] Chiticariu, L., Tan, W.C.: Debugging Schema Mappings with Routes. In: VLDB. pp. 79–90 (2006)
[6] Chomicki, J., Imieliński, T.: Finite representation of infinite query answers. ACM Transactions on Database Systems (TODS) 18(2), 181–223 (1993)
[7] Drabent, L., Nadjm-Tehrani, S.: Algorithmic Debugging with Assertions. In: Meta-programming in Logic Programming (1989)
[8] Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS (2007)
[9] Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update Exchange with Mappings and Provenance. In: VLDB. pp. 675–686 (2007)
[10] Kunen, K.: Declarative Semantics of Logic Programming. Bulletin of the EATCS 44, 147–167 (1991)
[11] Lausen, G., Ludäscher, B., May, W.: Transactions and Change in Logic Databases, chap. On Active Deductive Databases: The Statelog Approach, pp. 69–106. LNCS 1472 (1998)
[12] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (TOCL) 7(3), 499–562 (2006)
[13] Ludäscher, B.: Integration of Active and Deductive Database Rules. Ph.D. thesis, Albert-Ludwigs Universität, Freiburg, Germany (1998)
[14] Marczak, W.R., Huang, S.S., Bravenboer, M., Sherr, M., Loo, B.T., Aref, M.: SecureBlox: Customizable Secure Distributed Data Processing. In: SIGMOD (2010)
[15] Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. Logic Programming and Nonmonotonic Reasoning pp. 134–147 (2011)
[16] Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Workshop on Software Engineering for Answer Set Programming (SEA) (2007)
[17] Ramakrishnan, R., Ullman, J.: A Survey of Deductive Database Systems. Journal of Logic Programming 23(2), 125–149 (1995)
[18] Shapiro, E.: Algorithmic program debugging. Dissertation Abstracts International Part B: Science and Engineering, 43(5) (1982)
[19] Tobermann, G., Beckstein, C.: What's in a trace: The box model revisited. In: Automated and Algorithmic Debugging, LNCS, vol. 749, pp. 171–187 (1993)
[20] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. CoRR abs/1011.5332 (2010)
[21] Zaniolo, C., Arni, N., Ong, K.: Negation and Aggregates in Recursive rules: the LDL++ Aprrpach. Deductive and Object-Oriented Databases pp. 204–221 (1993)
[22] Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT Symposium on Foundations of Software Engineering (2002)