Interactive Debugging of Datalog Programs

ANDRÉ PACAK, JGU Mainz, Germany
SEBASTIAN ERDWEG, JGU Mainz, Germany

Datalog is used for complex programming tasks nowadays, consisting of numerous inter-dependent predicates. But Datalog lacks interactive debugging techniques that support the stepwise execution and inspection of the execution state. In this paper, we propose interactive debugging of Datalog programs following a top-down evaluation strategy called recursive query/subquery. While the recursive query/subquery approach is well-known in the literature, we are the first to provide a complete programming-language semantics based on it. Specifically, we develop the first small-step operational semantics for top-down Datalog, where subqueries occur as nested intermediate terms. The small-step semantics forms the basis of step-into interactions in the debugger. Moreover, we show how step-over interactions can be realized efficiently based on a hybrid Datalog semantics that adds a bottom-up database to our top-down operational semantics. We implemented a debugger for core Datalog following these semantics and explain how to adopt it for debugging the frontend languages of Soufflé and IncA. Our evaluation shows that our hybrid Datalog semantics can be used to debug real-world Datalog programs with realistic workloads.

CCS Concepts: • **Theory of computation** → **Constraint and logic programming**; **Operational semantics**; **Logic and databases**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: debugging, Datalog, small-step operational semantics, QSQR

ACM Reference Format:
André Pacak and Sebastian Erdweg. 2023. Interactive Debugging of Datalog Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 248 (October 2023), 28 pages. https://doi.org/10.1145/3622824

1 INTRODUCTION

Datalog is a logic programming language that was invented in the 1980s as a recursive query language for databases [Maier et al. 2018]. However, this is not how Datalog is being used nowadays. In the last 20 years or so, it has become increasingly popular to use Datalog as a programming language to solve all kinds of problems, from program analysis [Bravenboer and Smaragdakis 2009; Madsen et al. 2016; Szabó et al. 2021] to distributed computing [Abiteboul et al. 2005] and network monitoring [Alvaro et al. 2010, 2011]. Usages of Datalog usually have two things in common. First, they process graph-structured data of considerable size, which is well supported by Datalog's fixpoint semantics. Second, they involve complex Datalog programs, consisting of many inter-dependent rules. For example, even the simplest analysis from the Doop framework for Java bytecode consists of 560 lines of Datalog code that query 32 relations from the database and compute 78 inter-dependent derived relations [Bravenboer and Smaragdakis 2009]. Sophisticated Doop analyses are an order of magnitude larger still.

Unfortunately, development tools for Datalog have not been able to keep up with the growing size and complexity of Datalog programs. Specifically, there is no interactive debugging support available for Datalog to date. Interactive debuggers assist developers in program understanding

Authors' addresses: André Pacak, pacak@uni-mainz.de, JGU Mainz, Germany; Sebastian Erdweg, erdweg@uni-mainz.de, JGU Mainz, Germany.

2475-1421/2023/10-ART248
https://doi.org/10.1145/3622824

and bug finding by executing a program stepwise and exposing the program's execution state to the developer after each step. Interactive debugging support for Datalog is necessary to support modern Datalog programming. Specifically, there is an increasing number of Datalog programs that involve intricate "control flow" to capture the user's mental model of what happens when. A debugger should enable users to follow the intended control flow of such Datalog programs to trace the program's behavior, as in the following scenarios:

- Type checkers written in Datalog [Bembenek et al. 2020; Pacak et al. 2020]. Type checkers traverse the syntax tree and the debugger should retain that traversal order. Users need to be able to see the typing context during execution, as well as the set of candidate types in case of local ambiguity.
- Symbolic evaluators [Bembenek et al. 2020] and abstract interpreters [Pacak and Erdweg 2022]. Evaluators and interpreters execute (sub-)programs according to the evaluation order of the interpreted language. The debugger must retain that order. The execution state contains the current bindings of variables to values, which are relevant when debugging such interpreters.
- Flow-sensitive data-flow analysis as found in IncA [Szabó et al. 2018, 2021] and Doop [Bravenboer and Smaragdakis 2009]. A data-flow analysis propagates data-flow facts along the CFG of the analyzed program, either in forward or backward direction. A forward analysis traverses the CFG from function/main entry to function/main exit, a backward analysis is the other way around. The debugger must present the execution in that same order. What is more, data-flow analyses compute a fixpoint and the debugger must enable inspection of the execution state during each fixpoint iteration, not just the final result.

There are existing debugging tools for Datalog, but they treat Datalog as a query language for databases. In particular, existing debugging techniques for Datalog use post-mortem debugging and are based on provenance: They explain why a tuple was derived by computing the tuple's derivation tree. For example, consider the standard Datalog program that finds the paths of a directed graph:

$$\begin{array}{llll}
\mathsf{edge}(1,2). & \mathsf{edge}(2,3). & \mathsf{edge}(3,1). & \\
\mathsf{path}(X,Y) & :- \ \mathsf{edge}(X,Y). & & \text{// rule R1} \\
\mathsf{path}(X,Y) & :- \ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). & & \text{// rule R2}
\end{array}$$

Given tuple $\mathsf{path}(3,3)$, provenance-based approaches will give the following derivation tree as explanation of the derivation of the tuple:

$$\mathrm{R2}\ \frac{\mathsf{edge}(3,1) \qquad \mathrm{R2}\ \dfrac{\mathsf{edge}(1,2) \qquad \mathrm{R1}\ \dfrac{\mathsf{edge}(2,3)}{\mathsf{path}(2,3)}}{\mathsf{path}(1,3)}}{\mathsf{path}(3,3)}$$

Such visualizations can be useful in understanding a Datalog program, in particular for debugging data-driven Datalog programs without control flow. But why does provenance-based debugging not enable adequate debugging of the examples above? Provenance-based debugging does not follow the program's execution trace, but instead follows data-dependencies by using derivation trees. This induces the following shortcomings:

(1) The derivation tree only shows why a tuple was derived and does not show the execution state when a specific tuple was derived.
(2) Provenance-based debugging can only answer questions about ground tuples. For example, we cannot ask $\mathsf{path}(3,Y)$: what paths start at node 3 and why?

(3) The derivation tree shows a single derivation, but does not allow exploration of alternative (sub-)derivations that succeeded, nor of alternative (sub-)derivations that failed. That is, the derivation tree does not represent the execution trace of a Datalog program.

This paper develops interactive debugging for Datalog (with negation and constructors) that allows users to step through a Datalog execution and explore the intermediate execution state. To do so, we must first define what an execution trace for Datalog is. We argue that the standard bottom-up semantics of Datalog is ill-suited for debugging, since it results in a trace that follows the data rather than the structure of the program. Instead, we base our debugger on Datalog's top-down semantics, which starts with a query and steps in and out of rules until all query results have been found. In the literature, this semantics is known as the query/subquery approach, for which iterative and recursive algorithms exist [Vieille 1986]. To the best of our knowledge, we present the first small-step formulation of Datalog's top-down semantics, which forms the foundation of stepwise execution in our debugger.

One disadvantage of using the top-down semantics for debugging is its inefficiency in practice [Ullman 1989]. This is not much of an issue for individual step-into interactions, but when a user triggers a step-over or resume interaction, the debugger must run many steps in sequence and the inefficiency becomes a show-stopper. For example, consider a taint analysis implemented in Datalog that constructs a data-flow graph and then propagates taint along its edges. When debugging the taint analysis, a user may want to step over the data-flow graph construction and inspect the taint propagation only. Indeed, one of our experiments confirmed that the top-down semantics is too inefficient to evaluate even a simple Doop points-to analysis on a medium-sized Java codebase (timeout after a 10 minutes). This is why almost all Datalog solvers in practice rely on the bottom-up semantics, which can run the same analysis in less than 30 seconds. Technically, the bottom-up semantics is more efficient because it can compute n-ary joins for subqueries, whereas the top-down semantics must execute them in order and perform many binary joins instead. How can we make our debugger scale to real-world Datalog programs nonetheless?

We propose a novel hybrid debugging semantics for Datalog that mixes top-down and bottom-up evaluation. While stepping through the program trace, the user follows the top-down reduction steps. But, when stepping over a predicate call, we rely on the bottom-up semantics to compute the result of the skipped code. In fact, we can run the bottom-up semantics *once* prior to debugging and use the resulting database to provide the results for any number of stepped-over predicates. We further show how this approach can be extended to support stepping over recursive predicate calls, which may only produce a partial result in accordance with the current recursion depth. To this end, we exploit an incremental bottom-up semantics to temporarily "forget" tuples that will only be derived in later iterations.

We implemented two interactive Datalog debuggers, respectively based on the top-down and hybrid semantics. We evaluate the step-into and step-over performance of the debuggers on the path program and on an inter-procedural Java points-to analysis from Doop for realistic debugging scenarios. Our measurements show that the step-into performance is good, but the step-over interaction is a bottleneck in the top-down-only semantics. But using our hybrid semantics, the Datalog debugger scales to realistic workloads. We make the following contributions:

- We present the first formulation of a small-step operational semantics for Datalog. Our semantics corresponds to the well-known recursive query/subquery algorithm (Section 3).
- We present a novel hybrid semantics for Datalog that mixes top-down and bottom-up evaluation. A debugger can use top-down reductions for step-into and bottom-up results for step-over interactions (Section 4).

- We extend the hybrid semantics to allow step-over of recursive predicate calls through incremental maintenance of a logical relation between the two semantics (Section 5).
- We have implemented a debugger for core Datalog and show it can be used to debug languages that compile to core Datalog (Section 6).
- We evaluate the performance of our debugging approach on an inter-procedural points-to analysis on realistic workloads (Section 7).

2 WHY WE NEED INTERACTIVE DEBUGGING FOR DATALOG

This paper proposes an interactive debugging approach for Datalog programs, where users can explore and guide the execution of Datalog code. Our goal is to provide a debugger interface for Datalog that mirrors debuggers from imperative programming languages, featuring step into, step over, step out, resume, and breakpoints. In contrast, today's state-of-the-art Datalog debuggers support post-mortem debugging, where users can inspect derivation trees after tuples have been derived. This technique is known as provenance-based debugging. In this section, we discuss why provenance-based debugging is not sufficient for Datalog and why we need interactive debugging support for Datalog instead.

Why not to use derivation trees. Consider an encoding of type checking in Datalog as described by Pacak et al. [2020]. They encode the typing relation with two Datalog relations:

$$\begin{aligned}\texttt{typeOf} &\subseteq \text{Exp} \times \text{Type}\\ \texttt{lookup} &\subseteq \text{Exp} \times \text{Name} \times \text{Type}\end{aligned}$$

Relation `typeOf` assigns types to expressions, but does not carry a typing context. Instead, `typeOf` relies on `lookup` to resolve variable references, and `lookup` proceeds in reverse environment-passing style: it starts at the reference and walks up the tree until it finds a corresponding declaration using the auxiliary relation `parent` between AST nodes. This avoids the construction of typing contexts at Datalog run time, which has performance benefits [Pacak et al. 2020].

We consider a buggy Datalog program consisting of the rules for `typeOf` shown below:

$$\begin{array}{lll}
\texttt{typeOf}(e, T) & :- & \texttt{var}(e, x), \texttt{lookup}(e, x, T).\\
\texttt{typeOf}(e, \texttt{Bool}) & :- & \texttt{bool}(e, \_).\\
\texttt{typeOf}(e, \texttt{Nat}) & :- & \texttt{num}(e, \_).\\
\texttt{typeOf}(e, \texttt{Nat}) & :- & \texttt{add}(e, e1, e2), \texttt{typeOf}(e1, \texttt{Nat}), \texttt{typeOf}(e2, \texttt{Nat}).\\
\texttt{typeOf}(e, T \longrightarrow T') & :- & \texttt{lam}(e, x, T, b), \texttt{typeOf}(b, T').\\
\texttt{typeOf}(e, T') & :- & \texttt{app}(e, e1, e2), \texttt{typeOf}(e1, T \longrightarrow T'), \texttt{typeOf}(e2, \texttt{Nat}).
\end{array}$$

The relations `var`, `bool`, `num`, `add`, `lam`, and `app` are extensional relations encoding the input program. Note that this Datalog program constructs and destructs data in the form of types such as `Nat`, `Bool` and function types $T \longrightarrow T$. This is unproblematic as long as the constructed data is bound by extensional relations. We omit the encoding of the `lookup` relation for brevity.

We injected a bug into the typing rules above and will reveal it shortly. But first, consider the following term, which is accepted by the above rules even though it is not well-typed:

$$\lambda f : \texttt{Bool} \longrightarrow \texttt{Nat}.\ \lambda g : \texttt{Nat} \longrightarrow \texttt{Nat}.\ \lambda x : \texttt{Nat}.\ g\ ((f\ x) + 1)$$

We can try to debug our Datalog program using provenance-based debugging, which allows us to inspect the derivation tree. We need to inspect each node until we find an invalid derivation step, which indicates a buggy Datalog rule. We encourage the reader to try to find the buggy Datalog rule using the derivation tree before reading further.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{lookup}(\lambda g : \mathsf{N} \to \mathsf{N}.\ \ldots, g, \mathsf{N} \to \mathsf{N})}}{\texttt{lookup}(\lambda x : \mathsf{N}.\ \ldots, g, \mathsf{N} \to \mathsf{N})}}{\texttt{lookup}(g\ ((f\ x) + 1), g, \mathsf{N} \to \mathsf{N})}}{\texttt{lookup}(g, g, \mathsf{N} \to \mathsf{N})}}{\texttt{typeOf}(g, \mathsf{N} \to \mathsf{N})} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{lookup}(\lambda f : \mathsf{B} \to \mathsf{N}.\ \ldots, f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}(\lambda g : \mathsf{N} \to \mathsf{N}.\ \ldots, f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}(\lambda x : \mathsf{N}.\ \ldots, f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}(g\ ((f\ x) + 1), f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}((f\ x) + 1, f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}(f\ x, f, \mathsf{B} \to \mathsf{N})}}{\texttt{lookup}(f, f, \mathsf{B} \to \mathsf{N})}}{\texttt{typeOf}(f, \mathsf{B} \to \mathsf{N})} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{lookup}(\lambda x : \mathsf{N}.\ \ldots, x, \mathsf{N})}}{\texttt{lookup}(g\ ((f\ x) + 1), x, \mathsf{N})}}{\texttt{lookup}((f\ x) + 1, x, \mathsf{N})}}{\texttt{lookup}(f\ x, x, \mathsf{N})}}{\texttt{lookup}(x, x, \mathsf{N})}}{\texttt{typeOf}(x, \mathsf{N})}}{\texttt{typeOf}(f\ x, \mathsf{N})} \quad \dfrac{}{\texttt{typeOf}(1, \mathsf{N})}}{\texttt{typeOf}((f\ x) + 1, \mathsf{N})}}{\texttt{typeOf}(g\ ((f\ x) + 1), \mathsf{N})}}{\texttt{typeOf}(\lambda x : \mathsf{N}.\ g\ ((f\ x) + 1), \mathsf{N} \to \mathsf{N})}}{\texttt{typeOf}(\lambda g : \mathsf{N} \to \mathsf{N}.\ \lambda x : \mathsf{N}.\ g\ ((f\ x) + 1), (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N})}}{\texttt{typeOf}(\lambda f : \mathsf{B} \to \mathsf{N}.\ \lambda g : \mathsf{N} \to \mathsf{N}.\ \lambda x : \mathsf{N}.\ g\ ((f\ x) + 1), (\mathsf{B} \to \mathsf{N}) \to (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N})}$$

We abbreviate the derivation tree. For example, we do not show the queries of extensional relations such as `lam` and `app`. We also abbreviate `Nat` with `N` and `Bool` with `B`. We highlight some statistics about the complete tree:

$$\begin{array}{lcl} \texttt{typeOf}\ \text{derivations} & = & 10 \\ \texttt{lookup}\ \text{derivations} & = & 16 \\ \text{extensional derivations} & = & 29 \end{array}$$

Even for this small expression, the derivation tree is quite large and unwieldy containing 55 nodes. This makes debugging Datalog programs using provenance-based approaches unwieldy. That is, navigating and inspecting derivation trees is clunky.

The buggy Datalog rule is the one that handles typing function applications. The rule requires the argument to have type `Nat`, independent of the function's parameter type. Using provenance-based debugging, it can be very difficult to find such invalid rules. In particular, considering that Datalog programs are usually applied to large amounts of data (e.g., analyzing the entire JDK), yielding derivation trees that are many orders of magnitude larger. But this is not the only reason we need interactive Datalog debugging.

Our buggy Datalog rule for applications not only permits ill-typed terms, it also prohibits well-typed ones. Consider the following well-typed program p, for which `typeOf` fails to provide a derivation tree:

$$p = (\lambda f : \texttt{Nat} \to \texttt{Nat}.\ \lambda y : \texttt{Nat}.\ f\ y)(\lambda x : \texttt{Nat}.x + 1)$$

Again, the problem is that the rule for applications requires function arguments of type `Nat`, which is not the case here. Consequentially, there is no derivation tree that provenance-based debugging could provide to the user.

Why we need interactive debugging. Interactive debugging provides a well-known debugging interface to Datalog developers: breakpoints, resume, and stepping. Interactive debugging also allows developers to inspect the internal state of the Datalog program, such as the bindings of logical variables. The starting point of an interactive debugging session is a query that the developer would like to be answered, much like a unit test. And it does not matter whether the query is derivable or not: The user observes the progress of the Datalog solver interactively until it completes or fails.

Let's use interactive debugging on our Datalog typing rules using the same input program p from above. To find the bug, we set a breakpoint at the beginning of the application rule because we suspect an issue here. We start the session given the query $\mathtt{typeOf}(p, T)$. The first breakpoint we reach is for the outermost application with the following bindings:

$$e1 \mapsto \lambda f : \mathtt{Nat} \rightarrow \mathtt{Nat}.\, \lambda y : \mathtt{Nat}.\, f\ y \qquad e2 \mapsto \lambda x : \mathtt{Nat}.\, x + 1$$

We resume the computation and we reach the breakpoint again for the innermost application $f\ y$. We can step over the first typeOf atom or inspect its derivation using step into. After this atom, we obtain the bindings:

$$e1 \mapsto f \qquad e2 \mapsto y \qquad T \mapsto \mathtt{Nat} \qquad T' \mapsto \mathtt{Nat}$$

We can step into the second typeOf call to observe how lookup validates that variable y indeed has type Nat. Since all atoms of the application succeeded, we obtain a typeOf tuple for $f\ y$. We step out until we reach the rule processing the outermost application again, yielding the bindings:

$$\begin{array}{lcl} e1 & \mapsto & \lambda f : \mathtt{Nat} \rightarrow \mathtt{Nat}.\, \lambda y : \mathtt{Nat}.\, f\ y \\ e2 & \mapsto & \lambda x : \mathtt{Nat}.\, x + 1 \\ T & \mapsto & \mathtt{Nat} \\ T' & \mapsto & \mathtt{Nat} \end{array}$$

We step into the second typeOf call, which yields a query $\mathtt{typeOf}(\lambda x : \mathtt{Nat}.\, x + 1, \mathtt{Nat})$. By stepping further, we can observe that none of the typeOf rules can derive this query. In particular, the rule for lambda abstractions fails because the requested type Nat is not a function type. With this information, the user can discover the bug in the application rule, which should not have requested type Nat for arguments unconditionally.

Interactive debugging enables inspecting each step while determining if a tuple is derivable. In contrast, derivation trees only show the final result. This shows why interactive debugging is necessary: To manage the complexity of derivations using breakpoints, resume, step over, and step into. And to trace the logical reasoning, whether it succeeded or failed to derive a tuple.

3 SMALL-STEP OPERATIONAL SEMANTICS FOR TOP-DOWN DATALOG

In this paper, we propose to use top-down evaluation as a debugging semantics for Datalog programs. The top-down semantics is well-suited for debugging because it starts with a Datalog query issued by the user. Given a query, the top-down semantics recursively explores the rules of the Datalog program to satisfy the query and to derive the matching tuples. That is, the top-down semantics is goal-directed, while the bottom-up semantics is data-driven and populates tables eagerly. Since both semantics yield the exact same result [Green et al. 2013], the top-down semantics can safely be used for debugging even for systems that implement the bottom-up semantics.

The basis of our small-step semantics is the standard recursive query/subquery approach (QSQR) developed by Vieille [1986]. While this approach is well-documented in the literature [Abiteboul et al. 1995], its original formulation was soon found to be incomplete [Nejdl 1987; Vieille 1987]. The first complete QSQR semantics that provably finds all derivable tuples was proposed much later by Madalinska-Bugaj and Nguyen [2008]. Our semantics follows Madalinska-Bugaj and Nguyen and, in particular, their Remark 3.2 explaining that active queries can be maintained in a call stack.

In this section, we present the first small-step top-down Datalog semantics. Prior top-down Datalog semantics are ill-suited for debugging for three reasons. First, existing QSQR algorithms apply rules in a single big step, rather than stepping through their bodies. Second, they apply many rules simultaneously to compute a complete predicate, rather than considering one rule at a time to trace a predicate's growth. Third, they are presented as pseudo code that leaves many technical details implicit, such as how to modify tables using relational algebra. In contrast, we reformulate QSQR as a small-step operational semantics that makes all details explicit. This semantics will not

$$\begin{array}{lll}
\text{(Datalog programs)} & D & ::= \overline{r}, \overline{f} \\
\text{(rules)} & r & ::= p(\overline{X}) :- \overline{a}. \\
\text{(atoms)} & a & ::= p^s(\overline{t}) \mid \text{edb } p^s(\overline{t}) \mid t = t \mid t \neq t \\
\text{(signs)} & s & ::= + \mid - \\
\text{(terms)} & t & ::= c \mid X \\
\text{(facts)} & f & ::= p(\overline{c}). \\
\text{(constants)} & c & \\
\text{(variables)} & X &
\end{array}$$

Fig. 1. Abstract syntax of Datalog's surface language.

$$\begin{array}{lll}
\text{(value tables)} & v & ::= \text{table}(\overline{X}, \overline{T}) \\
\text{(tuples)} & T & ::= \overline{c} \\
 & & \\
\text{(rules)} & r & ::= \dots \mid v \\
\text{(atoms)} & a & ::= \dots \mid Q \\
\text{(queries)} & Q & ::= \mathbf{sq}(p^s(\overline{t}), v, v, v, r \vee \dots \vee r) \mid v^s
\end{array}$$

Fig. 2. Abstract syntax of Datalog's intermediate terms.

only serve as the basis for our debugger, but represents an important semantic artifact that will help substantiate programming-language research on Datalog.

3.1 Datalog Abstract Syntax

Before we discuss the small-step operational semantics, we introduce the abstract syntax of Datalog's surface language formally in Figure 1. A Datalog program consists of a collection of Datalog rules $\overline{r}$ describing the intensional database (IDB) and collection of facts $\overline{f}$ such as edge$(1, 2)$. describing the extensional database (EDB). We assume that the EDB is finite. A rule $p(\overline{X}) :- \overline{a}.$ consists of a rule head $p(\overline{X})$ and a rule body $\overline{a}$. The rule head $p(\overline{X})$ names the predicate the rule belongs to and declares columns, whose bindings will determine the derived tuples. The rule body is a sequence of atoms. An atom is either an (intensional) predicate call $p^s(\overline{t})$, extensional predicate call edb $p^s(\overline{t})$, an equality constraint $t = t'$, or an inequality constraint $t \neq t'$. We consider Datalog with stratified negation in this paper, which is why predicate calls have sign annotations s: A positive sign (+) indicates a regular predicate call, whereas a negative sign (−) indicates a negated call. Terms that occur inside atoms are either a constant value or a variable.

Note that a rule head only ranges over columns and not constants. Additionally, we only allow linear patterns in rule heads (no duplicate variable names) and assume rules belonging to the same predicate use the same column names. We make these assumptions without loss of generality as we can easily normalize arbitrary rules as the following example illustrates:

$$\begin{array}{lcl}
p(X, 1) :- \overline{a_1}. & & p(X, Y) :- \overline{a_1}, Y = 1. \\
p(X, A) :- \overline{a_2}. & \rightarrow & p(X, Y) :- \overline{a_2}, Y = A. \\
p(X, X) :- \overline{a_3}. & & p(X, Y) :- \overline{a_3}, Y = X.
\end{array}$$

The abstract syntax describes the surface language of Datalog. To formalize the small-step operational semantics of Datalog, we extend the abstract syntax with intermediate terms that only occur during evaluation and are not available to Datalog programmers. We summarize the required intermediate terms in Figure 2. First, rules evaluate to value tables v, which we add as an alternative to r. A value table table$(\overline{X}, \overline{T})$ consists of a list of column names $\overline{X}$ and a sequence of tuples $\overline{T}$,

Reduction Relations:		**Global Information:**	
(rule reduction)	$v \vdash r \;\longrightarrow^{R}\; r \dashv v$	(extensional database)	$EDB \in p \to v$
(atom reduction)	$v \vdash a \;\longrightarrow^{A}\; Q$	(intensional database)	$IDB \in p \to v$
(query reduction)	$Q \;\longrightarrow^{Q}\; Q$	(active queries)	$\Gamma \in p_\alpha \to \overline{v}$

Fig. 3. Reduction relations for top-down Datalog and the global state they interact with.

each of which is a sequence of constants. For readability, we usually denote the content of a table as a set of actual tuples, like $\{(1,2),(2,3)\}$. A table is only well-formed if the column names are mutually different and the number of columns matches the arity of all contained tuples. Note that $\text{table}(\overline{X},\emptyset)$ represents the empty table for any columns $\overline{X}$, whereas $\text{table}(\epsilon,\{()\})$ represents the unit table: a table without columns but with a single row, namely the empty tuple. The empty table is an absorbing element for natural joins and represents a failing computation. In contrast, the unit table is the neutral element for natural joins, which we use to represent computations that do not add bindings to the current environment:

$$\begin{array}{rcccl} \text{table}(\overline{X},\emptyset) \bowtie v & = & \text{table}(\overline{X},\emptyset) & = & v \bowtie \text{table}(\overline{X},\emptyset) \\ \text{table}(\epsilon,\{()\}) \bowtie v & = & v & = & v \bowtie \text{table}(\epsilon,\{()\}) \end{array}$$

The most interesting intermediate term is the one for subqueries. Subqueries Q occur when a predicate call $p^s(\overline{t})$ is reduced, which spawns a new subquery. Following the recursive query/-subquery approach QSQR, a subquery runs until it reaches a fixpoint. This may require multiple fixpoint iterations, which is why the subquery must manage quite a bit of auxiliary state. Specifically, a subquery $\mathbf{sq}(p^s(\overline{t}), v_a, v_r, v_{sup}, r_1 \vee \ldots \vee r_n)$ consists of five components:

(1) The original predicate call $p^s(\overline{t})$ to compute the bindings of free variables in $\overline{t}$ once the subquery reaches a fixpoint and terminates,
(2) the value table v_a that captures the arguments of the subquery,
(3) the subquery result v_r for the rules already evaluated,
(4) the supplementary table v_{sup} for the intermediate result of currently evaluating rule, and
(5) the current and remaining rules $r_1 \vee \ldots \vee r_n$.

The role of each component will become clearer when we discuss the semantics of subqueries. Eventually, a subquery evaluates to a value table v^s that carries the sign of the original predicate call. Note that all value tables are implicitly positive whenever we omit the sign annotation.

3.2 Reduction Relations and Global State

We model the small-step operational semantics of top-down Datalog through three mutually recursive reduction relations. Figure 3 shows the signatures of the three relations. Rule reduction $v_{sup} \vdash r \;\longrightarrow^{R}\; r' \dashv v'_{sup}$ of a rule r reduces and eventually eliminates atoms from the rule body. That is, we rewrite rules until all atoms are satisfied and consumed. Rule reduction happens under a supplementary table v_{sup} and produces a new supplementary table v'_{sup}. Supplementary tables are a standard element of top-down Datalog evaluation and roughly correspond to environments in other programming-language semantics. For example, a rule $\text{edge}(X,Y) :- X = 1, Y = 2.$ may start with a unit supplementary table, then reduce to $\text{edge}(X,Y) :- Y = 2.$ with a supplementary $\text{table}(X,\{(1)\})$, then to $\text{edge}(X,Y) :- \,.$ with $\text{table}(X,Y,\{(1,2)\})$. In contrast, to environments, the supplementary table may capture many possible values for a column and evaluation may filter entries that do not satisfy the rule body. For example, the evaluation of rule $\text{edge}(X,Y) :- X = 1, Y = 2.$ with initial supplementary $\text{table}(Y,\{(2),(3),(4)\})$ lists three alternative values for Y. The rule then reduces to $\mathsf{edge}(X, Y) :- Y = 2.$ with $\mathsf{table}(X, Y, \{(1, 2), (1, 3), (1, 4)\})$, and then to $\mathsf{edge}(X, Y) :- .$ with $\mathsf{table}(X, Y, \{(1, 2)\})$ by filtering rows that do not satisfy $Y = 2$.

Rule reduction depends on atom reduction $v_{sup} \vdash a \ \rightarrow^{A} \ Q$ of a under the supplementary table v_{sup}. An atom either reduces to an updated supplementary table or produces a new subquery. For subqueries, query reduction $Q \ \rightarrow^{Q} \ Q'$ is responsible for their execution and fixpoint iteration.

Our semantics uses three pieces of global state, which is shared between all reduction rules and also shown in Figure 3. We decided to model this state outside the intermediate terms to obtain a more concise semantics. Alternatively, the global state could be threaded in standard state-passing style. Function *EDB* collects all pre-defined facts $\overline{f}$ of the current Datalog program and represents the extensional database. For each extensional predicate p, *EDB* maps p to a value table v. Function *IDB* collects all tuples derived during the execution of a Datalog program and represents the intensional database. Initially, *IDB* maps all predicates to the empty table with columns matching the predicate's signature. During execution, *IDB* grows monotonically until we reach a fixpoint.

While *EDB* and *IDB* are common to all Datalog semantics, our top-down semantics also must track the active queries Γ. Essentially, Γ maps predicates to value tables that represent active argument values. The idea is to skip active arguments in recursive calls to prevent infinite recursion, while the fixpoint loop still ensures all derivable tuples are found. Technically, Γ must distinguish between different calls based on which parameters are bound at the call site, since the corresponding value tables have different columns. We employ adornments to distinguish bound columns b from free columns f as is standard for top-down Datalog, but also known from the magic set transformation [Beeri and Ramakrishnan 1991]. For example, the call $\mathsf{path}(1, 3)$ binds two parameters through adornment bb (bound and bound), but the adornment of call $\mathsf{path}(A, B)$ depends on the boundedness of A and B at the call site. Given a predicate p and an adornment α for each parameter of p, function Γ maps p_α to a stack of currently active query arguments.

3.3 Reduction Rules

We now present the reduction rules of the three reduction relations shown in Figure 4.

Rule reduction. There are three reduction rules defining the rule reduction relation: R-Step, R-Merge, and R-Result. R-Step is a simple congruence rule that reduces the next atom a of the current rule under the current supplementary table v_{sup}. This way, R-Step iteratively normalizes the frontmost atom, but it does not change the supplementary table.

Rule R-Merge applies when the first atom of a Datalog rule has been reduced to a value table v^s. We remove the value table v^s from the Datalog rule and merge it into the supplementary table v_{sup} using a helper function *merge* defined as follows:

$$merge(v_1, v_2^s) = \begin{cases} v_1 \bowtie v_2, & \text{if } s = + \\ v_1 \rhd v_2, & \text{if } s = - \end{cases}$$

This function combines two value tables based on the sign of the second operand. If the right-hand value table carries a positive sign (i.e., it stems from a positive predicate call), we merge the tables using a natural join. This effectively extends the current supplementary table with bindings for those variables that were free in the call of the predicate. In contrast, if the right-hand value table stems from a negative predicate call and carries a negative sign, we merge the tables using an anti-join. An anti-join $v_1 \rhd v_2$ only retains those rows of v_1 that do *not* have a match in v_2. In particular, an anti-join does not add any bindings to the supplementary table, since all Datalog variables must be bound by positive calls or positive equations.

$$\text{R-Step}\quad \frac{v_{sup} \vdash a \ \longrightarrow^{A}\ Q}{v_{sup} \vdash p(\overline{X}) :- a, \overline{as}. \ \longrightarrow^{R}\ p(\overline{X}) :- Q, \overline{as}. \dashv v_{sup}}$$

$$\text{R-Merge}\quad \frac{}{v_{sup} \vdash p(\overline{X}) :- v^{s}, \overline{as}. \ \longrightarrow^{R}\ p(\overline{X}) :- \overline{as}. \dashv \mathit{merge}(v_{sup}, v^{s})}$$

$$\text{R-Result}\quad \frac{}{v_{sup} \vdash p(\overline{X}) :- \epsilon. \ \longrightarrow^{R}\ \Pi_{\overline{X}}(v_{sup}) \dashv v_{sup}}$$

$$\text{A-Eq}\quad \frac{\mathit{unfree}(t, v_{sup}) \qquad \mathit{unfree}(t', v_{sup})}{v_{sup} \vdash t = t' \ \longrightarrow^{A}\ \sigma_{t=t'}(v_{sup})} \qquad \text{A-Neq}\quad \frac{\mathit{unfree}(t, v_{sup}) \qquad \mathit{unfree}(t', v_{sup})}{v_{sup} \vdash t \neq t' \ \longrightarrow^{A}\ \sigma_{t\neq t'}(v_{sup})}$$

$$\text{A-Eq-L}\quad \frac{X \notin \mathit{cols}(v_{sup}) \qquad \mathit{unfree}(t, v_{sup})}{v_{sup} \vdash X = t \ \longrightarrow^{A}\ \mathit{bind}(X, t, v)} \qquad \text{A-Eq-R}\quad \frac{X \notin \mathit{cols}(v_{sup}) \qquad \mathit{unfree}(t, v_{sup})}{v_{sup} \vdash t = X \ \longrightarrow^{A}\ \mathit{bind}(X, t, v)}$$

$$\text{A-EDB}\quad \frac{v_a = \mathit{eval}(\mathit{cols}(p), \overline{t}, v_{sup})}{v_{sup} \vdash \mathsf{edb}\ p^{s}(\overline{t}) \ \longrightarrow^{A}\ \rho_{\overline{t}/\mathit{cols}(p)}(\mathit{EDB}(p) \bowtie v_a)^{s}}$$

$$\text{A-Into}\quad \frac{v_a = \mathit{eval}(\mathit{cols}(p), \overline{t}, v_{sup}) \qquad (\Gamma', v_a') = \mathit{pushQuery}(\Gamma, p, v_a) \qquad v_a' \text{ not empty}}{v_{sup} \vdash p^{s}(\overline{t}) \ \longrightarrow^{A}\ \mathbf{sq}(p^{s}(\overline{t}), v_a', \mathrm{table}(\mathit{cols}(p), \emptyset), v_a', \mathit{rules}(p))}\ \Gamma := \Gamma'$$

$$\text{A-Skip}\quad \frac{v_a = \mathit{eval}(\mathit{cols}(p), \overline{t}, v_{sup}) \qquad (\Gamma', v_a') = \mathit{pushQuery}(\Gamma, p, v_a) \qquad v_a' \text{ empty}}{v_{sup} \vdash p^{s}(\overline{t}) \ \longrightarrow^{A}\ \rho_{\overline{t}/\mathit{cols}(p)}(\mathit{IDB}(p) \bowtie v_a)^{s}}$$

$$\text{A-Step}\quad \frac{Q \ \longrightarrow^{Q}\ Q'}{v_{sup} \vdash Q \ \longrightarrow^{A}\ Q'}$$

$$\text{Q-Step}\quad \frac{v_{sup} \vdash r \ \longrightarrow^{R}\ r' \dashv v_{sup}'}{\mathbf{sq}(p^{s}(\overline{t}), v_q, v_r, v_{sup}, r \vee \overline{rs}) \ \longrightarrow^{Q}\ \mathbf{sq}(p^{s}(\overline{t}), v_q, v_r, v_{sup}', r' \vee \overline{rs})}$$

$$\text{Q-Union}\quad \frac{}{\mathbf{sq}(p^{s}(\overline{t}), v_q, v_r, v_{sup}, v \vee \overline{rs}) \ \longrightarrow^{Q}\ \mathbf{sq}(p^{s}(\overline{t}), v_q, v_r \cup v, v_q, \overline{rs})}$$

$$\text{Q-Stable}\quad \frac{v_r \subseteq \mathit{IDB}(p) \qquad (\Gamma', v_a) = \mathit{popQuery}(\Gamma, p, v_q)}{\mathbf{sq}(p^{s}(\overline{t}), v_q, v_r, v_{sup}, \epsilon) \ \longrightarrow^{Q}\ \rho_{\overline{t}/\mathit{cols}(p)}(\mathit{IDB}(p) \bowtie v_a)^{s}}\ \Gamma := \Gamma'$$

$$\text{Q-Iterate}\quad \frac{v_r \nsubseteq \mathit{IDB}(p) \qquad \mathit{IDB}' = \mathit{IDB} \uplus p \mapsto (\mathit{IDB}(p) \cup v_r)}{\mathbf{sq}(p^{s}(\overline{t}), v_q, v_r, v_{sup}, \epsilon) \ \longrightarrow^{Q}\ \mathbf{sq}(p^{s}(\overline{t}), v_q, \mathrm{table}(\mathit{cols}(p), \emptyset), v_q, \mathit{rules}(p))}\ \mathit{IDB} := \mathit{IDB}'$$

Fig. 4. Reduction rules for rule reduction, atom reduction, and query reduction.

At last, when the rule body is empty, R-Result replaces the rule by the value table it produces. We obtain a rule's value table by projecting the final supplementary table according to the parameters in the rule's head.

Atom reduction. The atom reduction relation handles equalities and predicate calls. The rules A-Eq and A-Neq handle equality and inequality atoms whenever both operands are unfree. A term is unfree with respect to a table v if the term is a constant c or it is a variable X and $X \in \mathit{cols}(v)$. For unfree terms, A-Eq and A-Neq filter the current supplementary table v_{sup} according to the their constraint. This filtered supplementary table later replaces v_{sup} in the next R-Merge step since $v_{sup} \bowtie \sigma_f(v_{sup}) = \sigma_f(v_{sup})$ for any f. An actual implementation can of course avoid the extra join.

In addition, we define two rules A-Eq-L and A-Eq-R for the equality atom when one term is unfree but the other term is a free variable. These rules bind the free variable in the current supplementary table using the helper function *bind*:

$$\mathit{bind}(X, t, v) = \begin{cases} v \bowtie \text{table}(X, Y, \{(c, c) \mid c \in \Pi_Y(v)\}), & \text{if } X \notin \mathit{cols}(v), t = Y, Y \in \mathit{cols}(v) \\ v \bowtie \text{table}(X, \{(c)\}), & \text{if } X \notin \mathit{cols}(v), t = c \end{cases}$$

The first case is only triggered if the term is a variable Y bound by table v. Effectively, we extend table v with a new column X that is a copy of column Y. The second case is only triggered if the term is a constant. In this case, we extend the table v with a new column X that contains c in each row. Note that A-Eq-R is the same as A-Eq-L but where the right operand is an unbound variable.

The next three rules handle different kinds of predicate calls. Reduction rule A-EDB handles queries against the extensional database *EDB*. To this end, we first compute the argument table v_a of the call using a helper function *eval* to bind the columns of p to the argument values in $\overline{t}$:

$$\mathit{eval}(X_1, \ldots, X_n, t_1, \ldots, t_n, v) = \widetilde{\mathit{eval}}(X_1, t_1, v) \times \ldots \times \widetilde{\mathit{eval}}(X_n, t_n, v)$$

$$\widetilde{\mathit{eval}}(X, t, v) = \begin{cases} \text{table}(X, \{(c)\}), & \text{if } t = c \\ \rho_{X/Y}(\Pi_Y(v)), & \text{if } t = Y \end{cases}$$

Function *eval* takes a sequence of columns, a sequence of terms, and a value table. It calls another helper function $\widetilde{\mathit{eval}}$ on each column-term pair and builds the Cartesian product of their results. For a constant c, $\widetilde{\mathit{eval}}$ yields a singleton table binding column X to constant c. For a variable Y, $\widetilde{\mathit{eval}}$ instead yields a table with one column X bound to values of Y in v. With the Cartesian product of all of these tables in *eval*, we collect the induced bindings. Note that $\Pi_Y(v)$ is empty if Y is not bound in v, so that the resulting table may have some or all columns $X_1, \ldots, X_n$.

Based on the resulting argument table, rule A-EDB can find the matching tuples in *EDB* using a natural join, yielding a subset of $\mathit{EDB}(p)$. We then need to rename the columns to match the argument terms $\overline{t}$ of the predicate call, dropping columns that are constant in $\overline{t}$. We also superimpose the sign s of the call, which the subsequent R-Merge will take into account to update the supplementary table accordingly.

Reduction rule A-Into is only applicable if the query produced by $p^s(\overline{t})$ explores new argument tuples. It is important to track the previously visited argument tuples to ensure termination in the presence of recursive programs. For example, the Datalog program introduced in the introduction features a recursively defined predicate path, which could lead to the infinite call chain $\mathtt{path}(1, X)$, $\mathtt{path}(1, X)$, etc. when there is $\mathtt{edge}(1, 1)$. Note that the idea of distinguishing new from previous argument tuples is part of the query/subquery algorithm [Vieille 1986]; the contribution of this section is to reformulate this algorithm as a small-step operational semantics. We add a new query of p with argument table v_a to the global state Γ using helper function *pushQuery*:

$$\begin{aligned} \mathit{pushQuery}(\Gamma, p, v) &= (\Gamma', v') \\ \text{where} \qquad \alpha &= \mathit{adorn}(p, v) \\ v_1, \dots, v_n &= \Gamma(p_\alpha) \\ v' &= v \setminus (v_1 \cup \cdots \cup v_n) \\ \Gamma' &= \Gamma \uplus p_\alpha \mapsto v, v_1, \dots, v_n \end{aligned}$$

This function yields a filtered argument table v' that only contains new arguments (not already present on the stack) and an updated Γ' where we push v onto the stack associated with adorned predicate p_α. If the filtered argument table is non-empty, we create a new subquery with v'_a as argument table, an empty result table, and an initial supplementary table v'_a for the first rule of p.

The next reduction rule A-Skip complements A-Into and is only applicable if all argument tuples are already active . In this case, we do not recurse but read the tuples found so far from $\mathit{IDB}(p)$. We join this table with the argument table v_a to obtain only those tuples that match the current arguments. Finally, we have the congruence rule A-Step, which applies query reduction.

Query reduction. The query reduction relation takes care of subqueries that arise from predicate calls in A-Into. Rule Q-Step is a congruence rule that reduces the frontmost rule of the subquery using the rule reduction relation, updating the supplementary table of the subquery accordingly. Once a rule is normalized to a value table v, rule Q-Union adds the rule's result table v to the subquery result v_r. It then resets the supplementary table to the original argument table v_q so that the next rule in $\overline{rs}$ can continue.

The main responsibility of the query reduction relation is to iterate subqueries until all derivable tuples have been computed. To this end, rules Q-Stable and Q-Iterate handle complete subqueries that have no more rules to evaluate. If all tuples in the subquery result v_r have been derived before $v_r \subseteq \mathit{IDB}(p)$, we have found a fixpoint for the query of p with argument table v_q. We then remove the subquery's argument table v_q from Γ using the helper function *popQuery*:

$$\begin{aligned} \mathit{popQuery}(\Gamma, p, v) &= (\Gamma', v_1) \\ \text{where} \qquad \alpha &= \mathit{adorn}(p, v) \\ v_1, \dots, v_n &= \Gamma(p_\alpha) \\ \Gamma' &= \Gamma \uplus p_\alpha \mapsto v_2, \dots, v_n \end{aligned}$$

Removing v_q from Γ is necessary to allow exploring the subquery again later, when the intensional database *IDB* may have grown. Resetting Γ is the crucial fix to the QSQR algorithm proposed by Madalinska-Bugaj and Nguyen to guarantee completeness of the semantics. Since v_r did not contain new tuples, Q-Stable simply returns the tuples found in *IDB* for p and the original argument table v_a from the predicate call. If instead v_r contains new tuples $v_r \not\subseteq \mathit{IDB}(p)$, we must iterate the rules of p since we have not yet found a fixpoint. To this end, Q-Iterate adds the newly found tuples to *IDB* and re-initializes the subquery: discard the subquery result and restart with all rules of p. Since Datalog can only explore finitely many tuples, the iteration will terminate eventually when no new tuples are being derived.

3.4 Reduction Trace by Example

We have defined a complete small-step operational semantics for top-down Datalog. Here, we illustrate how the reduction semantics can be used to evaluate the example Datalog program from the introduction, but with additional edges:

$$\begin{array}{lll} \mathsf{edge}(1,2). & & \mathsf{edge}(2,3). \\ \mathsf{edge}(3,1). & & \mathsf{edge}(3,4). \\ \mathsf{path}(X,Y) & :- & \mathsf{edb}\ \mathsf{edge}(X,Y). \\ \mathsf{path}(X,Y) & :- & \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). \end{array}$$

Top-down evaluation always starts with a query. For example, consider the query $\mathsf{path}(1,Y)$ that finds all nodes Y reachable from 1. We initialize the intensional database map *IDB* as empty, and initialize active queries Γ with mapping $\mathsf{path}_{bf} \mapsto v_q$ where $v_q = \mathsf{table}(X, \{(1)\})$. Here and in subsequent examples of the paper, we omit positive signs from predicate calls and table values: these are implicitly positive. The extensional database *EDB* only contains the mapping

$$\mathsf{edge} \mapsto \mathsf{table}(X, Y, \{(1,2),(2,3),(3,1),(3,4)\})$$

We start with the following subquery:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, \emptyset, v_q, \begin{array}{l} \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Y). \\ \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). \end{array})$$

After a sequence of Q-Step applications for the first Datalog rule we arrive at the Datalog program where $v = \mathsf{table}(X, Y, \{(1,2)\})$. We reach the following intermediate term:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, \emptyset, v, v \vee \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).)$$

Now, Q-Union is applicable hence, we extend the subquery result to $v_r = \emptyset \cup v$, reset the supplementary table to v_q, and discard the first rule:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_q, \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).)$$

Again, after a sequence of Q-Step applications we get to $v_{sup} = \mathsf{table}(X, Z, \{(1,2)\})$:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) :- \mathsf{path}(Z,Y).)$$

A-Into extends Γ to $\mathsf{path}_{bf} \mapsto v'_q, v_q$ where $v'_q = \mathsf{table}(X, \{(2)\})$ and replaces the path call with a nested subquery Q:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) :- Q.)$$

$$Q = \mathbf{sq}(\mathsf{path}(Z,Y), v'_q, \emptyset, v'_q, \begin{array}{l} \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Y). \\ \mathsf{path}(X,Y) :- \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). \end{array})$$

While reducing the inner subquery Q, we will eventually encounter a predicate call with the same argument tuples again. At this point A-Skip is applicable and will force termination of the subquery. Hence, the inner subquery is replaced with table $v'' = \mathsf{table}(X, Y, \{(2,3),(2,4),(2,1),(2,2)\})$ to yield the intermediate term:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) :- v''.)$$

Now we apply R-Merge to yield $v'_{sup} = \mathsf{table}(X, Z, Y, \{(1,2,3),(1,2,4),(1,2,1),(1,2,2)\})$ followed by R-Result to produce the rule result:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v'_{sup}, \mathsf{table}(X, Y, \{(1,3),(1,4),(1,1),(1,2)\}))$$

Next, Q-Union extends the subquery result and discards the rule result:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, \mathsf{table}(X, Y, \{(1,2),(1,3),(1,4),(1,1)\}), v_q, \epsilon)$$

We derived new tuples hence, Q-Iterate is applicable and will extend the intensional database map *IDB*. The rule replaces the processed subquery with its initial version:

$$\mathbf{sq}(\mathsf{path}(1, Y), v_q, \emptyset, v_q, \begin{array}{l} \mathsf{path}(X, Y) :- \mathsf{edb}\ \mathsf{edge}(X, Y). \\ \mathsf{path}(X, Y) :- \mathsf{edb}\ \mathsf{edge}(X, Z), \mathsf{path}(Z, Y). \end{array})$$

At this point, the only active query is the initial query $\mathsf{path}(1, Y)$ as all other subqueries stabilized and therefore have been removed from the active queries map Γ by Q-Stable, hence Γ is $\mathsf{path}_{bf} \mapsto v_q$. Therefore, we will take the same steps as in the last iteration, producing the same subquery:

$$\mathbf{sq}(\mathsf{path}(1, Y), v_q, v_r', v_{sup}', \epsilon)$$

Hence, Q-Stable triggers as all tuples have already been discovered. The query has been successfully evaluated, there are no active subqueries as Γ is empty and a value table remains:

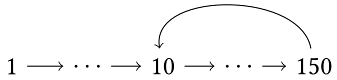
$$\mathsf{table}(X, Y, \{(1, 2), (1, 3), (1, 4), (1, 1)\})$$

Now we have shown a reduction trace for a concrete Datalog program given a starting query.

4 A HYBRID DATALOG SEMANTICS

The small-step operational semantics presented in the previous section can be used as a debugging semantics to follow the reduction trace of Datalog programs. The reduction rules describe the steps taken with the *step-into* interaction. This is not sufficient for debugging programs since developers also want to skip sub-computations using the *step-over* interaction. One way of implementing step-over of a predicate call is to step through the predicate using *step-into* until the predicate computation terminates, and show the predicate's result to the user. Unfortunately, this strategy has scalability issues for Datalog: The top-down debugging semantics is not fast enough to simulate predicate computations.

Consider we want to debug the `path` predicate over the following graph with 150 nodes:



We start the debugging session with query $\mathsf{path}(1, Y)$ and interactively use step-into until we reach query $\mathsf{path}(11, Y)$, which is first inner node of the cycle. To determine the result of $\mathsf{path}(11, Y)$ we want to use a step-over interaction. When we simulate the step-over interaction by repeatedly using step-into until the subquery terminates, it takes roughly 5 minutes to complete debugging of the program. Such long delays are unacceptable for a debugging interaction. Note that the simulated debugging trace consists of 255849 step-into interactions, which take 1-2 ms on average. We will discuss the scalability issue of step-into interactions in more detail in Section 7. The problem is not the step-into performance, but the simulation of step-over on top of step-into. Is there a better way to support step-over interactions for Datalog?

We propose to construct a hybrid Datalog semantics that uses top-down stepping for step-into but bottom-up results for step-over. As mentioned before, Datalog systems usually implement a bottom-up semantics, which is more efficient but ill-suited for debugging. The bottom-up semantics computes a database of all derivable tuples. The key idea is to read the predicate result from the bottom-up database when a step over occurs, rather than computing it on demand. Since the bottom-up database contains the tuples for all predicates in the program, we can effectively step over any predicate call in the program. Indeed, when using the bottom-up database to step over $\mathsf{path}(11, Y)$ in our example, the program completes debugging in only 180 ms. In this section, we formalize and exemplify step-over for non-recursive predicate calls, while Section 5 extends this idea to support recursive predicate calls.

4.1 Efficient Step-Over by Reading the Bottom-Up Database

We add the bottom-up database as global state *BU* and assume it has been computed prior to reduction. That is, when starting the debugger, a bottom-up Datalog solver must compute *BU*.

Global Information:

(bottom-up derived database) $\quad BU \in p \rightarrow v$

For each predicate in the debugged program, *BU* provides the final value table of derivable tuples for that predicate. For example, for the path predicate, *BU* contains all pairs (X, Y) of nodes for which X can reach Y. When stepping over a call $\mathtt{path}(1, Y)$, we can select all matching pairs from *BU* with a single join operation.

We formalize this behavior as an additional reduction rule A-Over for predicate calls:

$$\text{A-Over}\ \frac{\mathit{nonRecursiveCall}(p^s(\bar{t})) \qquad v_a = \mathit{eval}(\mathit{cols}(p), \bar{t}, v_{sup})}{v_{sup} \vdash p^s(\bar{t}) \ \rightarrow^A \ \rho_{\bar{t}/\mathit{cols}(p)}(BU(p) \bowtie v_a)^s}$$

Rule A-Over only applies to non-recursive predicate calls p (also not indirectly recursive) for reasons that we explain below. Technically, A-Over is applicable when the called predicate p is not in the currently active strongly connected component of the predicate call graph. We then determine the argument table v_a just like before in A-Into and A-Skip, but then look up the predicate result in *BU*. The resulting table carries the sign of the predicate call to mark if it was produced by a positive or negative call. We then merge the argument table v_a with the predicate result to obtain the tuples that match v_a according to this particular predicate call. Finally, we have to rename the columns of the table according to the argument terms like before.

Our new rule makes the semantics non-deterministic: predicate calls can be reduced with either one of the mutually exclusive A-Into or A-Skip, or alternatively with A-Over. This is ambiguity is by design, since users of the debugger should be able to choose how to continue evaluation. Thus, an important property of our extended semantics is confluence: All non-stuck reduction traces of a term normalize to the same value table. We provide a proof sketch for atom reduction.

Theorem 1 (Local Confluence for Atom Reduction). *Let $a \rightarrow^A b$ and $a \rightarrow^A c$, then there exists d such that $b \rightarrow^{A*} d$ and $c \rightarrow^{A*} d$.*

Proof sketch. There are only two interesting cases to consider, namely a is reduced by A-Skip and A-Over, or when a is reduced by A-Into and A-Over. In both cases $a = p^s(\bar{t})$

Case 1: Let $a \rightarrow^A b$ by A-Skip and $a \rightarrow^A c$ by A-Over. Then

$$b = \rho_{\bar{t}/\mathit{cols}(p)}(\mathit{IDB}(p) \bowtie v_a)^s \quad \text{and} \quad c = \rho_{\bar{t}/\mathit{cols}(p)}(BU(p) \bowtie v_a)^s.$$

Since the query/subquery approach and the bottom-up semantics compute the same result for the same query, $\mathit{IDB}(p) \bowtie v_a = BU(p) \bowtie v_a$ for stable p under argument table v_a. In A-Skip, p is stable and hence $b = c$.

Case 2: Let $a \rightarrow b$ by A-Into and $a \rightarrow c$ by A-Over. Then

$$b = \mathbf{sq}(p^s(\bar{t}), v'_a, \text{table}(\mathit{cols}(p), \emptyset), v'_a, \mathit{rules}(p)) \quad \text{and} \quad c = \rho_{\bar{t}/\mathit{cols}(p)}(BU(p) \bowtie v_a)^s.$$

Subquery b normalizes to $d = \rho_{\bar{t}/\mathit{cols}(p)}(\mathit{IDB}(p) \bowtie v_d)^s$ under $\rightarrow^Q$ using A-Step, where the last step is computed by Q-Stable. Since $v_d = v_a$ is the argument table pushed by A-Into and returned by *popQuery* in Q-Stable, we have $\mathit{IDB}(p) \bowtie v_d = BU(p) \bowtie v_a$ because the top-down and the bottom-up semantics coincide. Thus, $b \rightarrow^{A*} d = c$, which proofs confluence.

□

This proof sketch oversimplifies some concerns, in particular, the induction for the mutually recursive reduction relations and the base assumption that our small-step semantics for top-down Datalog is equivalent to a standard bottom-up semantics of Datalog. Resolving these concerns formally is far from trivial, given that the top-down and bottom-up semantics start evaluation from opposite sides (no shared control flow) and use contrary fixpoint strategies (depth-first in top-down, breadth-first in bottom-up). Mitigating these concerns will be a focus of our future work.

We want to highlight that even though the confluence property holds, repeat applications of A-Into do not necessarily simulate A-Over. Or conversely, a step over does not necessarily correspond to a sequence of step intos. Simply put, the difference is that A-Into modifies the intensional database *IDB* whereas A-Over does not. This can affect subsequent step-intos in a subtle way, as illustrated by the following scenario. If we step into a query $p(\bar{t})$, we need to determine the fixpoints for all predicate calls $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$ required to answer $p(\bar{t})$. Therefore, Q-Iterate is executed for each required subquery at least once, hence extending the intensional database *IDB*. In contrast, if we step over $p(\bar{t})$, we do not reach the calls $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$ and do not store their results in *IDB*. Therefore, when we later encounter another call of $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$, we may need extra fixpoint iterations to reach a stable result. Nonetheless, the fixpoint result of each subquery is the same, which is why confluence holds.

4.2 Reduction Trace by Example

To showcase A-Over, let us have a look at a reduction trace for the following Datalog program that derives parents and grandparents:

$$\begin{array}{rll} \mathsf{father}(Bob, Charlie). & \mathsf{father}(Bob, Dave). & \mathsf{father}(Dave, Mallory). \\ \mathsf{mother}(Alice, Charlie). & \mathsf{mother}(Eve, Dave). & \end{array}$$

$$\begin{array}{rcl} \mathsf{parent}(X, Y) & :- & \mathsf{edb\ father}(X, Y). \\ \mathsf{parent}(X, Y) & :- & \mathsf{edb\ mother}(X, Y). \\ \mathsf{grandparent}(X, Y) & :- & \mathsf{parent}(X, Z), \mathsf{parent}(Z, Y). \end{array}$$

We start the evaluation of the program with query $\mathsf{grandparent}(Bob, Y)$. Hence, we initialize *IDB* as empty and the active queries Γ with mapping $\mathsf{grandparent}_{bf} \mapsto v_q$ where $v_q = \mathsf{table}(X, \{(Bob)\})$. The initialized *EDB* contains the following mappings:

$$\begin{array}{lll} \mathsf{father} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory)\}) \\ \mathsf{mother} & \mapsto & \mathsf{table}(X, Y, \{(Alice, Charlie), (Eve, Dave)\}) \end{array}$$

For the hybrid semantics, we need to define the set of bottom-up derived tables *BU* as well. A bottom-up Datalog solver will derive the following set of tables:

$$\begin{array}{lll} \mathsf{father} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory)\}) \\ \mathsf{mother} & \mapsto & \mathsf{table}(X, Y, \{(Alice, Charlie), (Eve, Dave)\}) \\ \mathsf{parent} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory), \\ & & \qquad (Alice, Charlie), (Eve, Dave)\}) \\ \mathsf{grandparent} & \mapsto & \mathsf{table}((X, Y, \{(Bob, Mallory), (Eve, Mallory)\}) \end{array}$$

For query $\mathsf{grandprarent}(Bob, Y)$ we start evaluation with following subquery:

$$\mathbf{sq}(\mathsf{grandparent}(Bob, Y), v_q, \emptyset, v_q, \mathsf{grandparent}(X, Y) :- \mathsf{parent}(X, Z), \mathsf{parent}(Z, Y).)$$

Q-Step is applicable which applies R-Step which will reduce the first atom of the Datalog rule. The first atom is a predicate call of parent. We can either apply A-Into or A-Over because the call of parent is not a recursive one. We choose A-Over that utilizes *BU* to derive the table of the call. To produce the correct result for the predicate call, we join the bottom-up table *EDB*(parent) with

$v_a = \mathsf{table}(X, \{(Bob)\})$. At last, we need to rename the columns to match the arguments of the predicate call: Hence, A-Over replaces the atom with table

$$v = \mathsf{table}(X, Z, \{(Bob, Charlie), (Bob, Dave)\})$$

such that we obtain

$$\mathbf{sq}(\mathsf{grandparent}(Bob, Y), v_q, \emptyset, v_q, \mathsf{grandparent}(X, Y) :- v, \mathsf{parent}(Z, Y).)$$

This shows how we can utilize the bottom-up derived database to step-over a non-recursive predicate call instead of producing a new subquery that has to determine a fixpoint. Recursive predicate calls are more complicated as the next section explains.

5 A HYBRID SEMANTICS FOR RECURSIVE DATALOG

The previous section introduces the idea of a hybrid semantics that steps into according to top-down but steps over according to bottom-up. However, we limited this feature to non-recursive predicate calls. This conflicts with reality, where Datalog programs are highly recursive. Indeed, a Datalog debugger should allow stepping through the fixpoint computation of (mutually) recursive predicates, while also allowing to skip individual recursive sub-computations with step-over. However, supporting step-over for recursive predicate calls requires a more sophisticated semantics.

The problem is this: The bottom-up database contains the final result of a predicate. But when stepping over a recursive predicate call, we need to see a result that is consistent with the current progress within the fixpoint computation, not the final result. Consider the following example, where the directed graph contains two non-overlapping cycles: 1 ⟶ 2 ⟶ 3 ⟶ 1 and 4 ⟶ 5 ⟶ 4.

$$\begin{array}{lll}
\mathsf{edge}(1,2). & \mathsf{edge}(2,3). & \mathsf{edge}(3,1). \\
\mathsf{edge}(1,4). & \mathsf{edge}(4,5). & \mathsf{edge}(5,4). \\
\mathsf{path}(X,Y) & :- & \mathsf{edb}\ \mathsf{edge}(X,Y). \\
\mathsf{path}(X,Y) & :- & \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).
\end{array}$$

Assume we want to debug the computation of cycle 1 ⟶ 2 ⟶ 3 ⟶ 1, starting with query $\mathsf{path}(1, Y)$. Let's say we stepped to the second atom of the second rule. That is, the current program point is a subquery of $\mathsf{path}(2, Y)$ with

$$v_q = \mathsf{table}(X, \{(1)\}), \qquad v_r = \mathsf{table}(X, Y, \{(1, 2)\}), \qquad v_{sup} = \mathsf{table}(X, Z, \{(1, 2)\})$$

such that

$$\mathbf{sq}(\mathsf{path}(1, Y), v_q, v_r, v_{sup}, \mathsf{path}(X, Y) :- \mathsf{path}(Z, Y).).$$

If we now apply A-Over, we obtain all tuples of table path from the bottom-up database where column X is 2. The problem is that BU contains the final fixpoint of the derivation, which is inconsistent with our current debugging trace. Specifically, the bottom-up derived database BU is:

$$\begin{array}{lll}
\mathsf{edge} & \mapsto & \mathsf{table}(X, Y, \{(1,2), (1,4), (2,3), (3,1), (4,5), (5,4)\}) \\
\mathsf{path} & \mapsto & \mathsf{table}(X, Y, \{(1,2), (1,4), (2,3), (3,1), (4,5), (5,4), (1,3), (1,1), (1,5), (2,1), \\
 & & \qquad (2,2), (2,4), (2,5), (3,2), (3,3), (3,4), (3,5), (4,4), (5,5)\}
\end{array}$$

Hence, A-Over obtains $\mathsf{table}(X, Y, \{(2, 1), (2, 2), (2, 3), (2, 4), (2, 5)\})$. However, paths 1 ⟶ 2, 1 ⟶ 4, and 1 ⟶ 5 are only just being discovered and would not normally affect a recursive predicate call. Thus, A-Over should only yield $\mathsf{table}(X, Y, \{(2, 1), (2, 3)\})$: the tuples derivable for $\mathsf{path}(2, Y)$ in the current fixpoint iteration.

Fundamentally, the problem is that the bottom-up and top-down semantics employ conflicting fixpoint strategies. A bottom-up semantics computes the tuples of recursive predicates in iterations, where each iteration uses at least one newly derived tuple from the previous iteration (known as semi-naïve evaluation). In contrast, the recursive query/subquery approach of the top-down semantics follows the recursive predicate calls until a query is revisited. Thus, even if we were to track the iteration count of tuples in the bottom-up database, we would not know which tuples to produce when in the top-down semantics.

One correct solution for this problem would be go back to simulating step-over using step-into. We propose an alternative that exploits the bottom-up database, but ignores all tuples not derivable in the current fixpoint iteration.

5.1 Blacklisting Tuples That Are Ahead of Their Time

blacklisting tuples that are ahead of their time To support step-over for recursive predicate calls efficiently, we want to read the call result from the bottom-up database like before. But, as explained above, the bottom-up database contains tuples we need to ignore to produce a result that is consistent with the fixpoint computation of the top-down semantics. To this end, we must answer the following two questions:

(i) Which tuples do we need to ignore to obtain a correct result from the bottom-up database?
(ii) How can we realize this strategy efficiently?

To answer the first question, consider the following intermediate Datalog term:

$$\mathbf{sq}(\mathsf{path}(X,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) :\!-\ \mathbf{sq}(\mathsf{path}(Z,Y), v'_q, v'_r, v'_{sup}, \mathsf{path}(X,Y) :\!-\ \mathsf{path}(Z,Y).).)$$

When stepping over the inner predicate call $\mathsf{path}(Z,Y)$, which tuples do we need to ignore from the bottom-up database? Careful investigation of the query/subquery approach reveals the answer: We must ignore tuples that are currently being computed by active subqueries and all tuples that depend on them. For an active subquery of p, only tuples already written to $IDB(p)$ by Q-Iterate may be considered, but no other intermediate answers. For our example, we need to ignore tuples that depend on the tuples $\mathsf{path}(X,Y)$ under table v_q and $\mathsf{path}(Z,Y)$ under table v'_q. More concretely, when we start with query $\mathsf{path}(1,Y)$, we ignore all paths that begin at node 1 for the outer subquery. The inner subquery is of the form $\mathsf{path}(Z,Y)$ under supplementary table $\mathsf{table}(X,Z,\{(1,2)\})$, hence we also ignore all paths that begin at node 2. Hence, a step over the innermost $\mathsf{path}(Z,Y)$ under $\mathsf{table}(Z,\{(3)\})$ will produce $\mathsf{table}(Z,Y,\{(3,1)\})$ only, because paths starting at 1 are only visible in later fixpoint iterations.

Now the question arises how we can exploit the bottom-up database while ignoring specific tuples and their dependents. Clearly, we cannot compute a new bottom-up database whenever the set of active subqueries changes. But, what if we use an incremental bottom-up semantics for Datalog that can react to external changes, such as IncA [Szabó et al. 2021] or Differential Datalog [Ryzhyk and Budiu 2019]? Is it then possible to mark ignored tuples and have the incremental semantics update the bottom-up database appropriately without recomputing it from scratch?

We found a way to engineer the incremental bottom-up database to do exactly that. Our solution consists of two steps: Generate extensional *blacklist* predicates into the original Datalog program, and insert/remove ignored tuples into the *blacklist* predicates during debugging. For example, we rewrite the rules of our path predicate as follows:

$$\begin{aligned}
\mathsf{path}(X,Y) \ &:\!-\ \mathsf{edb\ bl\_path\_bf}^-(X), \mathsf{edb\ bl\_path\_fb}^-(Y), \mathsf{edb\ bl\_path\_bb}^-(X,Y), \mathsf{edb\ edge}(X,Y).\\
\mathsf{path}(X,Y) \ &:\!-\ \mathsf{edb\ bl\_path\_bf}^-(X), \mathsf{edb\ bl\_path\_fb}^-(Y), \mathsf{edb\ bl\_path\_bb}^-(X,Y), \mathsf{edb\ edge}(X,Z),\\
&\qquad \mathsf{path}(Z,Y).
\end{aligned}$$

For each adornment of path that occurs in the program, we add a blacklist guard to the path rules. Blacklist guards are negative calls that ensure a tuple is not blacklisted and thus may be derived.

The blacklist transformation is relatively simple and syntactically rewrites a Datalog program:

$$Blacklisted(p(\overline{t}) :\!-\ \overline{a}.) = \begin{cases} p(\overline{t}) :\!-\ \overline{a}. & \text{if } nonRecursive(p) \\ p(\overline{t}) :\!-\ \overline{a}, \overline{b}. & \text{if } recursive(p),\ \overline{\alpha} = adornments(p), \\ & \quad b_i = bl\_p\_\alpha_i^-(\pi_{\alpha_i}(\overline{t})) \end{cases}$$

For a non-recursive predicate (also not indirectly recursive), no changes are necessary since the predicate becomes stable in a single fixpoint iteration. However, for recursive predicates (also indirectly recursive), we introduce additional atoms $\overline{b}$, one for each adornment in $\overline{\alpha}$. Each new atom b_i is a negative call to a new extensional predicate named $bl\_p\_\alpha_i$, where p is the name of the current predicate and α_i is the adornment currently considered. The blacklist predicate is applied to those terms in $\overline{t}$ that are marked bound in the adornment α_i. The blacklist transformation only needs to be applied to the Datalog program evaluated bottom-up. The top-down evaluation can operate on the original Datalog program.

5.2 Formalizing Step-Over for Recursive Predicates

The previous subsection introduced blacklist predicates that prevent certain derivations in the bottom-up database. The blacklist predicates are all part of the extensional database, meaning their contents are specified manually. Initially, all blacklists are empty. When using an incremental Datalog solver, we can insert and remove tuples from extensional predicates dynamically, and the incremental solver updates all derived predicates accordingly. We make use of this feature to blacklist tuples that match active subqueries as part of the debugging semantics, as we explain here.

We extend our hybrid semantics from Section 4 to maintain and use blacklist predicates. To this end, we require global state to keep track of the blacklisted queries and the updated bottom-up database. Luckily, we already maintain global state to track active subqueries, namely Γ. Therefore, we only introduce new global state that represents the updated bottom-up database.

Global Information:

(blacklist-aware bottom-up database) $BU \in (p, \Gamma) \longrightarrow v$

Since the bottom-up database operates on the blacklist-transformed program, it only yields derived tuples not depending on active queries. We make this explicit by adding Γ as a parameter to *BU*.

We can now extend and adapt the semantics to support efficient step-over recursive predicate calls. We introduce a new reduction rule A-OverRecursive to step over recursive predicate calls:

$$\text{A-OverRecursive} \quad \frac{recursiveCall(p^s(\overline{t})) \qquad v_a = eval(cols(p), \overline{t}, v_{sup})}{v_{sup} \vdash p(\overline{t})^s \ \longrightarrow^A\ \rho_{\overline{t}/cols(p)}(BU(p,\Gamma) \bowtie v_a \ \cup\ IDB(p) \bowtie v_a)^s}$$

We determine the argument table v_a as in A-Over, but in A-OverRecursive we read from the blacklist-aware bottom-up database $BU(p, \Gamma)$. This yields the derivable tuples for the predicate call, except for tuples that depend on active subqueries. For the first fixpoint iteration of a subquery, this behavior is correct and sufficient. Only in later fixpoint iterations, it is important to also consider the tuples derived by the top-down debugger itself in $IDB(p)$. Therefore, A-OverRecursive yields the relevant bottom-up tuples and the tuples derived top-down.

With the help of the blacklist and an incremental bottom-up Datalog solver, we can now step over arbitrary predicate calls efficiently. Specifically, we do not need to simulate stepped-over predicates with step-into. We now have all the tools to implement a working debugger for Datalog.

6 DEBUGGER IMPLEMENTATION AND FRONTEND DEBUGGING

The small-step operational semantics presented in the previous sections are sufficient for the definition of an interactive Datalog debugger. We validate this claim by implementing a fully functional debugger that we make available open source[1] as part of the IncA Datalog framework [Pacak et al. 2022]. In this section, we describe the debugger implementation and optimizations we applied on top of the formal semantics. Since Datalog is often used as an intermediate representation (IR) rather than as a frontend language, we also show how the debugger can support debugging of languages that compile to Datalog.

6.1 From Small-Step Semantics to Debugger

We implemented a debugger for Datalog following the formal semantics presented in this paper. In particular, the debugger supports stepping through query execution with the following interactions:

- Step into: A single step of the query reduction relation $\rightarrow^{Q}$, using any rule but A-Over.
- Step over: Conversely, a single step of the query reduction relation, using any rule but A-Into.
- Step out: Abort and discard the current subquery and perform a step-over interaction of its call instead. Note how this avoids the repeated application of step-over until reaching the end of the current subquery.

The debugger uses IncA's incremental bottom-up Datalog solver [Szabó et al. 2021].

Our debugger implementation deviates from the formal semantics in multiple aspects. First of all, the debugger uses a flattened representation for the intermediate terms of Datalog: Rather than nesting active subqueries in the syntax, we maintain a call stack where each active subquery has its own call frame. This is possible because the semantics deterministically executes rules and atoms from left to right, so that there can be only one active subquery at each recursion level (otherwise would need to maintain a call tree). Rule A-Into pushes a new call frame to the stack (instead of creating a subquery) and rule Q-Stable replaces the current call frame with the subquery result. To pop from the call stack, we introduce a new rule Q-Result that pops the subquery result from the stack and replaces the predicate call of the current active subquery with the resulting atom table. This way, the call stack provides constant-time access to the most recently created subquery, to which all other reduction rules apply. In addition, the implementation groups the atom reduction rules that handle equalities within a single rule A-Eq. At last, the IR supports executing arbitrary Scala code. Hence, we extend the debugger with rule A-Prim that executes the Scala code with each supplementary table entry as input.

The debugger incorporates other optimizations that are more local, but also have a noticeable performance impact. First, rules with overlapping preconditions (like A-Into and A-Skip) are implemented such that the preconditions are run at most once. Second, we implement the termination condition in rules Q-Stable and Q-Iterate by comparing the size of the current result to the result size in the bottom-up database, which avoids a costly subset check in each iteration. We also only store derived tuples for recursive predicates in *IDB* as non-recursive predicates do not need fixpoint iteration. Third, we represent value tables as immutable b-trees with efficient table operations [Jordan et al. 2019]. In particular, this data type provides an efficient join implementation, which is crucial for executing Datalog on considerable amounts of data.

Finally, our debugger also supports breakpoints. A breakpoint identifies a predicate, a rule inside a predicate, or an atom inside a rule at which execution should stop. Breakpoints can be registered and de-registered with the debugger. Breakpoints affect the semantics of the step-over reduction: A-Over may not be applied to predicate calls that can transitively reach an active breakpoint. We

[1] https://gitlab.rlp.net/plmz/inca-scala

add this as an additional precondition. Similarly, a step-out interaction is not applicable if the remainder of the current subquery may reach a breakpoint. In both situations, we fall back to a new *resume* interaction instead:

- Resume: The resume interaction runs the program until it terminates or a breakpoint is hit. To run the program, resume iteratively applies prioritized interactions: 1. step out if possible, 2. step over if possible, 3. step into otherwise.

6.2 Debugging Datalog Frontends

Our debugger supports core Datalog with negation. However, Datalog dialects usually add numerous language features on top of core Datalog or provide entirely different frontend languages. For example, the Souffle language provides a component system, typed relations, and multi-head rules [Scholz et al. 2015]. And IncA not only provides a Datalog-flavored constraint language [Szabó et al. 2016], but also functional programming frontend called functional IncA that compiles to core Datalog [Pacak and Erdweg 2022]. In principle, we can use the core Datalog debugger to evaluate core Datalog code generated by Souffle and IncA. However, the user would see debugging steps in terms of the core Datalog code: core Datalog rules with auxiliary atoms and variables. This breaks the abstraction of the frontend language and hinders the applicability of the debugger severely. This subsection explains how we adopted our core Datalog debugger to debug Souffle and functional IncA code at their abstract level.

The basic idea for debugging Datalog frontends is simple: We define a partial function *lift* from core Datalog intermediate terms to the intermediate terms of the frontend language that should be displayed as steps to the user. We also define the inverse of *lift* called *lower*, which translates the frontend term back to its core Datalog original. Given a reduction relation $\rightarrow^Q$ of core Datalog, we can then define a reduction relation $\rightsquigarrow$ for the frontend language as follows:

$$\frac{\mathit{lower}(b) \rightarrow^Q Q_1 \rightarrow^Q \ldots \rightarrow^Q Q_{n+1} \qquad \mathit{lift}(Q_1) = \bot \quad \ldots \quad \mathit{lift}(Q_n) = \bot \qquad \mathit{lift}(Q_{n+1}) = b'}{b \rightsquigarrow b'}$$

That is, the frontend debugger steps in core Datalog using query reduction until it reaches an intermediate term that can be lifted into the frontend. That is, lifting a query Q is successful ($\mathit{lift}(Q) \neq \bot$). While this approach is not particularly novel, this paper provides the necessary formal infrastructure to explain and specify this technique precisely. For breakpoints, we define another lowering function that translates frontend breakpoints into the core Datalog breakpoints. The breakpoint-lowering function should be defined such that the lowered breakpoints stop at intermediate terms that can be lifted back into the frontend. In practice, we found that implementing the lifting and the two lowering functions was easy for Souffle and functional IncA.

The frontend reduction relation $\rightsquigarrow$ specified above works well when a Datalog frontend merely elaborates into core Datalog. That is, a single construct compiles to a single Datalog rule, as is the case with most Souffle features. Frontend languages that apply more complex compilation rules require extra care. For example, functional IncA features *if-then-else* expressions that compile into two rules: one rule that asserts the condition evaluates to true and executes the *then* branch, and one rule that asserts the condition evaluates to false and executes the *else* branch. When using the lifting approach of $\rightsquigarrow$ from above, we obtain an inadequate reduction relation for functional IncA:

- If the condition evaluates to true, we correctly step through the corresponding rule first. However, we will subsequently step through the *else* rule, re-evaluate the condition, and only then discard this rule as unsatisfiable (always failing).
- Conversely, if the condition fails, we step through the *then* rule until the condition failed, then step through the same prefix in the *else* rule, and only then step through the *else* branch.

Either way, we ran the condition and all atoms leading up to the condition twice, which is not what we expect from a debugger of a functional language. Therefore, we augment the frontend debugger with extra logic to skip unsatisfiable rules and satisfied prefixes. Unsatisfiable rules result from conditionals (and pattern matching) when execution has already committed to another branch (or match case). Satisfied prefixes also result from conditionals (and pattern matching) when execution reached a condition that failed and needs to try another branch (or match case). With these two mechanisms in place, we can adopt the core Datalog debugger to provide debugger for functional IncA that meets user expectations.

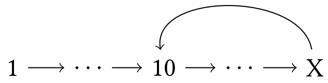
7 PERFORMANCE EVALUATION

In this section, we evaluate if the proposed top-down debugging approach can be used for realistic Datalog programs, and if the hybrid semantics is necessary. To this end, we measure the step-into and step-over running times for both semantics on synthetic and real-world Datalog programs. We performed all measurements on a machine with an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 13.2.1, Java 11.0.18.

7.1 Is a Hybrid Semantics Necessary?

Without our hybrid semantics, top-down Datalog debuggers must simulate step-over interactions through a sequence of step-into interactions. To evaluate if this is feasible, we measure their respective running times and compare them to the time required by a bottom-up Datalog solver.

Setup. We use the educational Datalog example that derives transitive paths between nodes of a graph, which we have seen throughout this paper. We consider the following synthetic graph, where we can configure the length of the cycle by setting node $X \in \{10, 20, \ldots, 990, 1000\}$:



We measure and compare three scenarios for the initial query path$(1, Y)$:

(1) Exclusively use step-into interactions until the query terminates.
(2) Use step-into interactions until reaching query path$(11, Y)$, then use step over. Since path$(11, Y)$ is within the graph's cycle, this exercises our A-OverRecursive from the hybrid semantics.
(3) Use a bottom-up semantics without stepping, as required to initialize the hybrid semantics.

Note that we chose to step over the path call on first inner node of the cycle as the blacklist-propagation will delete all paths that cross node 11. This scenario is the worst-case scenario which will stress the blacklist-propagation of the incremental bottom-up semantics when using A-OverRecursive. We execute these scenarios and measure their running times for $X \in \{10, 20, \ldots, 990, 1000\}$. We report the average times of 20 runs after discarding 5 warmup runs.

Results. Figure 5 shows the running times on the left and number of executed steps on the right. For the step-into scenario (1), we observe an exponential running time, whereas the step-over scenario (2) and bottom-up scenario (3) exercise quadratic running times. Interestingly, the number of steps remains constant when using a step over at path$(11, Y)$, since we skip the computation for remainder of the graph as X grows. Only the running time grows with X, because the incremental blacklist propagation has to delete a quadratic number of path tuples with increasing X.

The results show that using repeated step-into interactions to simulate a step-over interactions is infeasible as it requires too many steps, taking too much time. Our hybrid semantics and its step-over interaction based on an incremental bottom-up Datalog is necessary and effective.

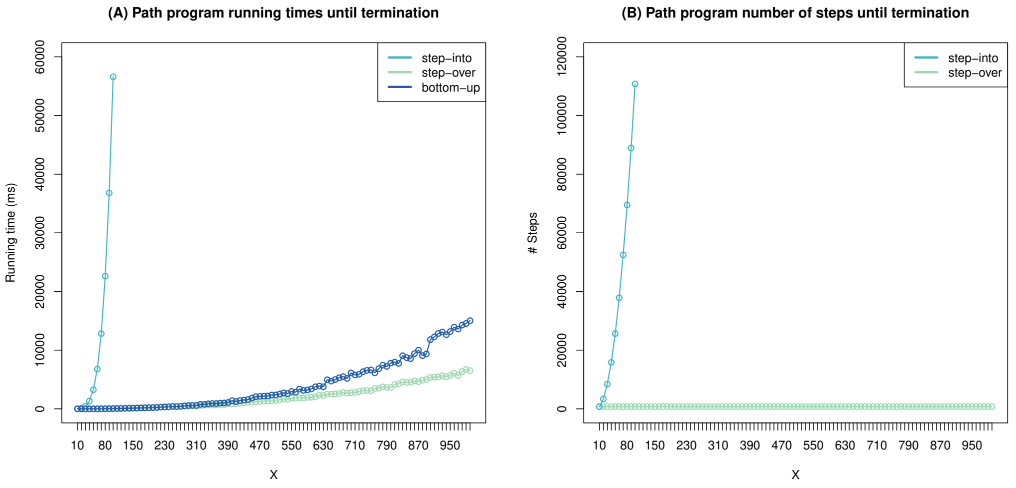


Fig. 5. Running times of step-into vs step-over vs bottom-up for path program

7.2 Real-World Workloads

The previously measured path example is an educational example but not a real-world program. Even for the educational example it is infeasible to only use step-into when debugging because it takes an excessive amount of steps. Hence, interactive debugging of real-world programs stands or falls with the responsiveness of the step-over interaction. To show that our hybrid top-down debugging approach is applicable to real-world programs with realistic workloads, we measure the step-over performance for an inter-procedural points-to analysis of JVM bytecode provided by the Doop framework [Bravenboer and Smaragdakis 2009].

Setup. We debug Doop's context and flow insensitive points-to analysis on the MiniJava compiler.[2] This program only consists of 6.5k lines of Java code, but the inter-procedural analysis also transitively analyzes all reachable parts of the JDK. We use the Doop fact extractor to translate the JVM bytecode to an EDB that describes the program and the transitively reachable parts of the JDK, amounting to 380 MB of data.

We consider three debugging scenarios for measuring the step-into performance:

(1) Derive all classes implementing method `accept` and show how they are computed.
(2) Derive all supertypes of two given JDK types and show how they are computed.
(3) Derive if a given variable points to a given heap location and follow the computation.

Additionally, we define three variations of scenario (3) for measuring the step-over performance:

(3a) Step into VarPointsTo, but step over all other predicate calls.
(3b) Step into VarPointsTo and StaticFieldsPointsTo, but step over all other predicate calls.
(3c) Step into VarPointsTo and InstanceFieldPointsTo, but step over all other predicate calls.
(3d) Step into VarPointsTo and Reachable, but step over all other predicate calls.

Note that VarPointsTo, StaticFieldPointsTo, InstanceFieldPointsTo, and Reachable are mutually recursive. We run each of these debugging scenarios only once, where we consider a timeout of 10 minutes after doing warmup with a timeout of 2 minutes 5 times. For scenarios (1–3), we measure

[2] https://github.com/mtache/minijavac

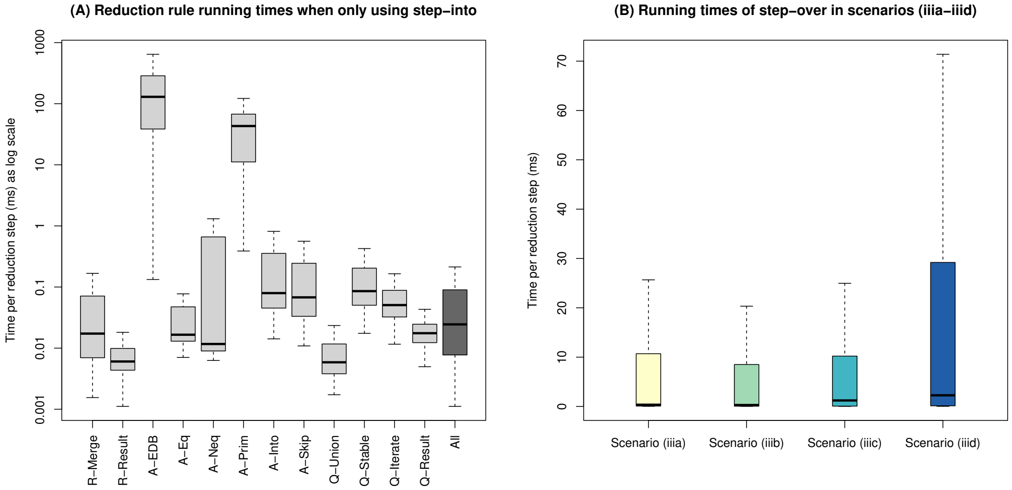


Fig. 6. Running times (ms) of reduction rules of Doop points-to analysis.

the running time of each individual reduction rule, whereas for scenarios (3a–3d) we only measure the running time of step-over interactions (A-Over or A-OverRecursive).

Results. We present the results of the benchmark in Figure 6. The boxplots on the left show the running times (log scale) of all reduction rules executed during scenarios (1–3), including an additional boxplot combining all 38029 reduction steps. Most reduction rules only require at most 1 ms to execute, except A-EDB (up to 1000 ms) and A-Prim (up to 100 ms). The boxplots also exclude outliers, but we want to mention that out of the 2202 steps with A-Into, there was one extreme outlier that took 40s and a few of outliers that took around 10s to execute. The running times of A-EDB and the outliers of A-Into are slow because they construct an index on-the-fly for a complete *EDB* or argument relation prior to joining, which can take a significant amount of time. With enough engineering effort, it should be possible to optimize the index construction and operations on tables to reduce these times considerably. A-Prim executes Scala code using reflection on each entry of a value table, which has a severe penalty in our current implementation.

We show the step-over running times (linear scale) for the interactive debugging scenarios (3a–3d) on the right-hand side of Figure 6. The mean step-over running time is below 10 ms and indeed almost all step-over steps require less than 100 ms. Again, the boxplots exclude outliers: Scenario (3c) has one outlier in 1278 step-over interactions that took 168s and scenario (3d) has two outliers in 888 step overs taking 62s and 40s. These outliers are due to the incremental deletion of tuples that have a high impact on the bottom-up database. Szabó et al. [2021] showed that such high-impact changes exist but are rare for static-analysis applications of Datalog; most changes have low impact and can be executed efficiently.

We conclude that interactive debugging of real-world Datalog programs is feasible with the hybrid top-down debugger. Both step-into and step-over interactions are executed fast enough for interactive debugging. Even though there are some extreme outliers, they are rare and do not dominate the debugging session. Thus, a hybrid debugging semantics is necessary (previous subsection) and sufficient (this subsection).

8 RELATED WORK

We propose to enable an interactive debugging experience for Datalog programs by exploring the top-down reduction trace of a Datalog program using a top-down evaluation strategy. To efficiently allow for step-over interactions, we provide a hybrid semantics for Datalog that mainly uses top-down but relies on a bottom-up derived database when stepping over predicate calls. We will compare our debugging approach of Datalog programs with related work in this section.

The explanation system of DeDEx utilizes derivation trees to show why specific tuples have been derived [Wieland 1990]. This system can only show derivation trees for tuples that have been derived. Hence, it cannot explain why a tuple has not been derived. Additionally, the system requires a complete tuple. In contrast, our approach allows to start with arbitrary queries of predicates. That is, we are not forced to only select tuples that have been derived and we can provide partial queries to explain how a specific query has been answered. This allows to show the reduction trace for multiple tuples. Their system allows for a restricted query facility mode. This mode allows to ignoring and adding rules and tuples when constructing derivation trees for predicates. Our approach does not allow for ignoring or adding specific rules. However, our hybrid approach allows for ignoring tuples to allow for stepping over cyclic predicate calls. Their approach will only ignore tuples and it will not effect tuples that depend on the existence of the ignored tuple. We propose to use an incremental Datalog solver to always keep a consistent deductive database as a basis.

Russo and Sancassani [1991] propose to explore an evaluation post-mortem, meaning after executing a Datalog program. They use a derivation tree to show why a tuples has been derived as well. They instruct their Datalog solver to derive derivation tree fragments during bottom-up evaluation. This allows for efficient exploration of derivation trees. We also rely on the tuples derived by a Datalog solver. However, our approach allows to use an off-the-shelf incremental Datalog solver instead of extending the Datalog solver to generate additional information.

Explain is capable of explaining how specific tuples were derived and which tuples are derived based on a specific tuple [Arora et al. 1993]. They use derivation trees to visualize how a specific tuple has been derived as well. Again, this system does not allow partial tuples which our approach is capable of. They also adapt the Datalog solver to derive fragments of the derivation tree. Again, our approach does not require extending the Datalog solver. Explain is capable of supporting aggregation. Our formalization does not consider aggregation, but we support it because the Datalog solver IncA we use supports recursive lattice-based aggregation [Szabó et al. 2018].

Caballero et al. [2008a,b] propose algorithmic debugging for Datalog programs. They derive a computation graph representing the evaluation of a Datalog query where the nodes represent predicate tables for a specific query and the edges determine which queries are needed to derive the predicate table. Their approach traverses the computation graph and ask the user if stored predicate table is valid to identify buggy vertices or buggy circuits. A node of the computation graph is buggy if the predicate table is non-valid while all predicate tables of the immediate descendants are valid. A buggy circuit is a cycle in the computation graph where all vertices are invalid. Köhler et al. [2012] propose a similar approach, that is declarative debugging, since they also derive a graph connecting tuples with rule firings. They use Statelog [Lausen et al. 1998] to record in which iteration and how often a tuple has been derived. Based on the graph it is possible to extract and explore subgraphs with pre-defined and ad hoc defined queries such as counting rule firings. In contrast, our approach does not follow algorithmic or declarative debugging. We propose to follow the computation model of a Datalog program instead. That is, we follow the execution trace of a top-down evaluation while utilizing a bottom-up derived database to answer step-overs efficiently.

Soufflé's debugging approach scales to large deductive databases [Zhao et al. 2020]. Soufflé introduce a new provenance lattice that stores derivation annotations including the rule deriving the tuple and what the minimal height of the derivation tree deriving the tuple is. Additionally, they describe a new bottom-up evaluation semantics that derives tuples of the provenance lattice. This enables them to construct derivation trees of minimal height in the presences of recursive Datalog programs. Their approach requires a complete tuple to construct a derivation tree. In contrast, our approach allows to explore the reduction trace of partial tuples. They also support answering the question of why a tuple was not derived but it requires user-interaction. However, it is a semi-automated approach where the user has to select the rule that should have derived the tuple and provide a substitution for free variables within the select rule. Our approach is completely automated in the sense that we just follow the reduction trace of an unsatisfiable query.

ViatraQuery uses slicing of a rete network to identify faulty nodes [Ujhelyi et al. 2016]. ViatraQuery is the Datalog solver we use for bottom-up evaluation. Their approach identifies a sub-network in addition to inputs that derive or do not derive a specific tuple. That enables to pinpoint errors in the rule definitions. Our approach also only explores Datalog predicates that contribute to answer the initial query. Because they identify a sub-network of the rete network, it is not easily possible to provide a mapping from the Datalog frontends to the sub-network and back.

9 CONCLUSION

Current Datalog debugging techniques are based on the view of Datalog as a query language for databases where they use provenance information and follow the data instead of the program execution. In recent years, Datalog has been used as a programming language instead. In this paper, we propose an interactive debugging technique for Datalog programs that follows a top-down evaluation strategy while exposing the execution state. We define the top-down evaluation strategy as a small-step operational semantics where each application corresponds to a step-into interaction. We base the small-step operational semantics on the evaluation strategy recursive query/subquery. While recursive query/subquery is not novel [Vieille 1986], we are the first to define a small-step operational semantics formulation of it. This semantic artifact will help substantiate programming-language research on Datalog. Top-down evaluation of Datalog is less efficient than bottom-up semantics, hence all state-of-the-art Datalog solvers use semi-naïve evaluation. Thus, simulating step-over using only step-into interactions is highly inefficient. We propose to define a hybrid semantics that combines top-down and bottom-up evaluation. To this end, we can access the bottom-up derived database when stepping over a predicate call. In particular, we use an incremental Datalog solver to allow for debugging fixpoint iteration efficiently. We implement the interactive debugger based on the small-step operational semantics within the IncA framework [Szabó et al. 2021]. Our implementation allows for setting breakpoints as well. We enable debugging frontends that compile to core Datalog such as Soufflé and functional IncA by lifting and lowering intermediate terms back and forth. In future work we want to explore how we can extend the interactive debugger with condition breakpoints to explore specific evaluation points more efficiently. The performance evaluation shows that top-down debugging of Datalog programs is feasible when using a hybrid top-down semantics to efficiently answer step-over interactions.

ACKNOWLEDGEMENTS

We thank Sebastian Mies, David Klopp, Runqing Xu, Prashant Kumar, and the anonymous reviewers for their valuable feedback. This work was funded in part by German Research Foundation (DFG)-451545561 and 508316729.

REFERENCES

Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. 2005. Diagnosis of asynchronous discrete event systems: datalog to the rescue!. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database*

Systems, June 13-15, 2005, Baltimore, Maryland, USA, Chen Li (Ed.). ACM, 358–367. https://doi.org/10.1145/1065167.1065214

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, 223–236. https://doi.org/10.1145/1755913.1755937

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. 1993. Explaining Program Execution in Deductive Systems. In *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93, Phoenix, Arizona, USA, December 6-8, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 760)*, Stefano Ceri, Katsumi Tanaka, and Shalom Tsur (Eds.). Springer, 101–119. https://doi.org/10.1007/3-540-57530-8_7

Catriel Beeri and Raghu Ramakrishnan. 1991. On the power of magic. *The Journal of Logic Programming* 10, 3 (1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q Special Issue: Database Logic Progamming.

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

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, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2008a. A New Proposal for Debugging Datalog Programs. *Electron. Notes Theor. Comput. Sci.* 216 (2008), 79–92. https://doi.org/10.1016/j.entcs.2008.06.035

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2008b. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *Semantics in Data and Knowledge Bases, Third International Workshop, SDKB 2008, Nantes, France, March 29, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4925)*, Klaus-Dieter Schewe and Bernhard Thalheim (Eds.). Springer, 143–159. https://doi.org/10.1007/978-3-540-88594-8_8

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (nov 2013), 105–195. https://doi.org/10.1561/1900000017

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 327–339. https://doi.org/10.1145/3293883.3295719

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative Datalog Debugging for Mere Mortals. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7494)*, Pablo Barceló and Reinhard Pichler (Eds.). Springer, 111–122. https://doi.org/10.1007/978-3-642-32925-8_12

Georg Lausen, Bertram Ludäscher, and Wolfgang May. 1998. On Active Deductive Databases: The Statelog Approach. In *Transactions and Change in Logic Databases, International Seminar on Logic Databases and the Meaning of Change, Schloss Dagstuhl, Germany, September 23-27, 1996 and ILPS '97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97) Port Jefferson, NY, USA, October 17, 1997, Invited Surveys and Selected Papers (Lecture Notes in Computer Science, Vol. 1472)*, Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov (Eds.). Springer, 69–106. https://doi.org/10.1007/BFb0055496

Ewa Madalinska-Bugaj and Linh Anh Nguyen. 2008. Generalizing the QSQR Evaluation Method for Horn Knowledge Bases. In *New Challenges in Applied Intelligence Technologies*, Ngoc Thanh Nguyen and Radoslaw P. Katarzyniak (Eds.). Studies in Computational Intelligence, Vol. 134. Springer, 145–154. https://doi.org/10.1007/978-3-540-79355-7_14

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM / Morgan & Claypool, 3–100. https://doi.org/10.1145/3191315.3191317

Wolfgang Nejdl. 1987. Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, Peter M. Stocker, William

Kent, and Peter Hammersley (Eds.). Morgan Kaufmann, 43–50. http://www.vldb.org/conf/1987/P043.PDF

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 127:1–127:28. https://doi.org/10.1145/3428195

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 20–32. https://doi.org/10.1145/3564719.3568686

Francesco Russo and Mirko Sancassani. 1991. A Declarative Debugging Environment for DATALOG. In *Logic Programming, First Russian Conference on Logic Programming, Irkutsk, Russia, September 14-18, 1990 - Second Russian Conference on Logic Programming, St. Petersburg, Russia, September 11-16, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 592)*, Andrei Voronkov (Ed.). Springer, 433–441. https://doi.org/10.1007/3-540-55460-2_32

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, 56–67. http://ceur-ws.org/Vol-2368/paper6.pdf

Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. 2015. A Datalog Source-to-Source Translator for Static Program Analysis: An Experience Report. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*. IEEE Computer Society, 28–37. https://doi.org/10.1109/ASWEC.2015.15

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

Zoltán Ujhelyi, Gábor Bergmann, and Dániel Varró. 2016. Rete Network Slicing for Model Queries. In *Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9761)*, Rachid Echahed and Mark Minas (Eds.). Springer, 137–152. https://doi.org/10.1007/978-3-319-40530-8_9

Jeffrey D. Ullman. 1989. Bottom-Up Beats Top-Down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, Avi Silberschatz (Ed.). ACM Press, 140–149. https://doi.org/10.1145/73721.73736

Laurent Vieille. 1986. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In *Expert Database Systems, Proceedings From the First International Conference*. Benjamin/Cummings, 253–267.

Laurent Vieille. 1987. A Database-Complete Proof Procedure Based on SLD-Resolution. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes)*, Jean-Louis Lassez (Ed.). MIT Press, 74–103.

Christian A. Wieland. 1990. Two Explanation Facilities for the Deductive Database Management System DeDEx. In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90), 8-10 October, 1990, Lausanne, Switzerland*, Hannu Kangassalo (Ed.). ER Institute, 189–203.

David Zhao, Pavle Subotic, 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. https://doi.org/10.1145/3379446

Received 2023-04-14; accepted 2023-08-27