Datalog with First-Class Facts

Thomas Gilray
Washington State University
thomas.gilray@wsu.edu

Arash Sahebolamri
Syracuse University
arash.sahebolamri@gmail.com

Yihao Sun
Syracuse University
ysun67@syr.edu

Sowmith Kunapaneni
Washington State University
sowmith.kunapaneni@wsu.edu

Sidharth Kumar
University of Illinois at Chicago
sidharth@uic.edu

Kristopher Micinski
Syracuse University
kkmicins@syr.edu

ABSTRACT

Datalog is a popular logic programming language for deductive reasoning tasks in a wide array of applications, including business analytics, program analysis, and ontological reasoning. However, Datalog's restriction to flat facts over atomic constants leads to challenges in working with tree-structured data, such as derivation trees or abstract syntax trees. To ameliorate Datalog's restrictions, popular extensions of Datalog support features such as existential quantification in rule heads (Datalog$^{\pm}$, Datalog$^{\exists}$) or algebraic data types (Soufflé). Unfortunately, these are imperfect solutions for reasoning over structured and recursive data types, with general existentials leading to complex implementations requiring unification, and ADTs unable to trigger rule evaluation and failing to support efficient indexing.

We present $\mathcal{DL}^{\exists!}$, a Datalog with first-class facts, wherein every fact is identified with a Skolem term unique to the fact. We show that this restriction offers an attractive price point for Datalog-based reasoning over tree-shaped data, demonstrating its application to databases, artificial intelligence, and programming languages. We implemented $\mathcal{DL}^{\exists!}$ as a system Slog, which leverages the uniqueness restriction of $\mathcal{DL}^{\exists!}$ to enable a communication-avoiding, massively-parallel implementation built on MPI. We show that Slog outperforms leading systems (Nemo, Vlog, RDFox, and Soufflé) on a variety of benchmarks, with the potential to scale to thousands of threads.

PVLDB Reference Format:
Thomas Gilray, Arash Sahebolamri, Yihao Sun, Sowmith Kunapaneni, Sidharth Kumar, and Kristopher Micinski. Datalog with First-Class Facts. PVLDB, 18(3): 651 - 665, 2024.
doi:10.14778/3712221.3712232

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/harp-lab/slog-lang1.

1 INTRODUCTION

Datalog has emerged as a de-facto standard for iterative reasoning applications such as business analytics [6, 26], context-sensitive program analysis [5, 14], and ontological query answering [71]. Datalog's success has been a harmony between the power of recursive queries and increasingly high-performance implementations, effectively leveraging parllelism on multiple cores [52, 85], clusters [88, 91], or even GPUs [92]. Modern implementations are efficiently operationalized via compilation to iterated joins over efficient data structures (e.g., tries [54], BTrees [53], or hash tables [44]) representing sets of tuples over word-sized values.

Unfortunately, while Datalog's syntactic restrictions lead to straightforward decidability results, those same restrictions make Datalog ill-suited to applications which reason inductively over tree-shaped data such as derivations or syntax. In this work, we introduce $\mathcal{DL}^{\exists!}$, an extension of Datalog to enable *first-class* facts. $\mathcal{DL}^{\exists!}$ takes inspiration from languages such as Datalog$^{\pm}$ and Datalog$^{\exists}$, which allow existential quantifiers in the head of rules [20]. $\mathcal{DL}^{\exists!}$ restricts Datalog$^{\exists}$ to a *uniqueness quantification* where every fact implies the existence of its own identity. Such a choice meaningfully enhances Datalog's expressivity—enabling a form of equality-generating dependency [10] over tree-shaped facts. In $\mathcal{DL}^{\exists!}$, every tuple is uniquely identified by a Skolem term unique to the fact. Such a choice rapidly enables computing over tree-shaped data, while yielding a lock-free, communication-avoiding implementation approach, compatible with state-of-the-art parallelizing implementation techniques for Datalog [45, 62].

We formalize $\mathcal{DL}^{\exists!}$ as the syntactic extension of Datalog to include *unique existential quantification* ($\exists!$) in the head of rules. This restriction ensures that every fact is identified by a Skolem term unique to the fact. We present a chase-based semantics for $\mathcal{DL}^{\exists!}$, whose implementation is significantly simplified by the restriction that all facts are structurally unique, sidestepping the challenges of unification by associating each (sub)-fact with a unique intern id. The syntactic restrictions of $\mathcal{DL}^{\exists!}$ forbid unifying the head with the body, thereby preventing cyclic reference and yielding finite expansion sets. While $\mathcal{DL}^{\exists!}$ is only semi-decidable in general, running subsequent iterations of a $\mathcal{DL}^{\exists!}$ query will always produce increasingly-larger facts, with each iteration adding finitely-many facts. Practically, $\mathcal{DL}^{\exists!}$'s properties fortuitously enable a trivially-parallel implementation as a simple extension of modern parallel relational algebra implementations [44, 45, 61].

Specifically, our contributions are as follows:

(1) We introduce $\mathcal{DL}^{\exists!}$, a Datalog with first-class facts, i.e., in which all facts are uniquely identified via a nested Skolem term. We present a semantics for $\mathcal{DL}^{\exists!}$ based on the restricted chase, and then present $\mathcal{DL}_{\mathcal{S}}$, a language which compiles down to $\mathcal{DL}^{\exists!}$; $\mathcal{DL}_{\mathcal{S}}$ enables more natural programming with directly-nested facts, equivalent in power, and is the basis for our implementation.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712232

(2) We present a series of applications demonstrating $\mathcal{DL}^{\exists!}$'s generality and relevance to a broad array of fields. Specifically, we discuss $\mathcal{DL}^{\exists!}$'s application to provenance, algebraic data (*e.g.*, as included in Soufflé), functional programming (as in IncA [77]), structural abstract interpretation [100], and type systems [78, 80].

(3) We implement Slog, a fully-featured, data-parallel engine, which compiles $\mathcal{DL}_{\mathcal{S}}$, with syntactic sugar, to a Message Passing Interface (MPI) based runtime. While there are no explicit constructs for data parallelism in $\mathcal{DL}_{\mathcal{S}}$, the restrictions of $\mathcal{DL}^{\exists!}$ enable leveraging recent work in (balanced) parallel relational algebra, BPRA, which uses hash-based distribution strategies and load-balancing, scaling to thousands of threads. We show that BPRA can be extended to support $\mathcal{DL}^{\exists!}$ with just a small change to its implementation: a trivially-parallelizable mechanism for tuple deduplication, to intern and assign unique references for new tuples (§4).

(4) We evaluate Slog on a series of applications in graph analysis with provenance and program analysis, both with and without materializing provenance information, illustrating how our framework supports a unified perspective on first-class facts as provenance, at most, or as unique enumerated references, at least. Operationally, our approach enables us to design rich, highly-parallel static analyses which leverage $\mathcal{DL}^{\exists!}$ as a foundation. We evaluate Slog by implementing these applications and performing a series of experiments showing scaling and comparisons against other state-of-art systems. These include (a) why-provenance comparisons versus Nemo [51], VLog [98], and RDFox [74], (b) context-sensitive analysis of Linux, and (c) results showing an exponential gap versus Soufflé when using algebraic data types (§5).

2 DATALOG WITH FIRST-CLASS FACTS

Positive Datalog ($\mathcal{DL}^{+}$) rules contain a head and body clauses:

$$H(x, y, \ldots) \leftarrow B(a, b, \ldots) \wedge \ldots$$

In $\mathcal{DL}^{+}$, the body contains only positive clauses, and a rule is *safe* whenever every variable in the head $(x, y, \ldots)$ is contained in the body; such a restriction ensures finite solutions. It is trivial to extend the definition of safety to stratified negation (i.e., $\mathcal{DL}$): every variable that occurs in a negated subgoal must also appear in a positive subgoal. Such safety conditions ensure that $\mathcal{DL}$ programs may be evaluated in a bottom-up manner, iterating an immediate consequence operator to a necessarily-finite fixed point.

Despite being a popular implementation platform for an array of applications, Datalog's restriction to flat rules over atomic constants has proven limiting in applications such as ontological reasoning. Such settings necessitate structurally representing and computing over knowledge, well beyond the power of Datalog's flat rules. In a response to this limitation, a growing array of languages build upon $\mathcal{DL}$ to enable reasoning over ontologies, including Datalog$^{\pm}$ and Datalog$^{\exists}$. In parallel, a growing breadth of work in programming languages compiles higher-order functional programs [77] and type systems [78] to Datalog, grappling with its semantic limitations via monomorphization or ad-hoc extensions.

2.1 $\mathcal{DL}^{\exists!}$: its syntax and chase semantics

The core of our contribution is to identify a price point for the implementation of high-performance, parallel Datalog engines which also effectively enables applications such as provenance, ontology reasoning, structural type systems, and functional programming. Specifically, we introduce $\mathcal{DL}^{\exists!}$, an extension of Datalog to enable first-class facts. $\mathcal{DL}^{\exists!}$ introduces a single syntactic extension to Datalog: every fact is assigned an *identity*, which may be otherwise used as any other base value: referenced (selected and joined upon) in any body clauses and used to populate any (non-*id*) column of the head clause. Rules in $\mathcal{DL}^{\exists!}$ take the form:

$$\left(\exists! id_H.\ id_H = H(x, y, \ldots)\right) \leftarrow id_B = B(a, b, \ldots) \wedge \ldots$$

The $\exists!$ in $\mathcal{DL}^{\exists!}$ refers to the fact that the identity of the deduced fact is *uniquely* determined for this rule and all of the values used in the head. Unlike Datalog$^{\pm}$ and Datalog$^{\exists}$, the logical variable quantified by $\exists$ does not appear as a column but instead serves as an annotation associated with the generated tuples. This design choice means that the identity of the head clause cannot be unified with logical variables in the body clauses or with other logical variables in the head. For example, the following two rules are not valid $\mathcal{DL}^{\exists!}$ queries:

Example 2.1. Ill-formed unification of annotations in $\mathcal{DL}^{\exists!}$:

$$\begin{array}{lcl} c = H(a, b) & \leftarrow & id = B(a, b, c) \\ \exists! c.H_1(a, b) \wedge H_2(a, c) & \leftarrow & B(a, b) \end{array}$$

By excluding unification between the identity of the head clause and logical variables in the body or head, we ensure that identity values are used in a strictly acyclic manner. This restriction prevents undesired non-termination issues that can arise from chasing cyclic dependencies. Meanwhile, forbidding identity unification across head clauses enables more parallel implementations of the $\mathcal{DL}^{\exists!}$ query engine. It simplifies the satisfaction check for existentially quantified queries to merely verifying whether the inferred head clause tuple already exists, thereby avoiding extra communication in distributed settings. We will elaborate on this in Section 4.1.

The use of the existential quantifier in rule heads extends Datalog with *tuple-generating dependencies* (TGDs), which trigger tuple generation based upon the insertion of other tuples in the database. While such an extension significantly enhances Datalog's expressivity, it comes at a cost: adding TGDs makes queries undecidable in general and leads to infinite results. As a response, there has been a significant amount of interest in tractable restrictions to TGDs which ensure decidability [20, 23, 63]. Much of this work leverages the *restricted chase* [21], a sound and complete algorithm for query answering over ontologies of disjunctive existential rules. The restricted chase discovers all derivations in bottom-up fashion, generating labeled-nulls and unifying Skolem terms on demand.

We give the semantics of $\mathcal{DL}^{\exists!}$ via a variant of the restricted chase. The function $chase(\Sigma, \mathcal{I})$ iteratively builds a frontier of newly-generated facts in Δ, a (potentially infinite) stream of facts; the iteration, i, is initially set to 0 and Δ^0 is set to $\mathcal{I}$, the input database. Algorithm 1 is then repeated until $\Delta^{i-1} = \emptyset$. The algorithm finds all possible matches θ in the current generation, and then checks whether a first-class fact $H(x, ...)$ exists in the database $\Delta^{[0,i]}$, substituting the head via the match θ. If not, the algorithm generates a fresh labeled null, building the updated substitution θ'; last, θ' is applied to the rule's head, $\left(id_H = H(x, ...)\right)$, to materialize the new first-class fact. Across iterations, Δ records increasingly-tall facts, equating each fresh null with a novel Skolem term.

Algorithm 1 applyRules(Σ, Δ, i)

$\Delta^{i+1} = \emptyset$
for all $\left(\exists! id_H.H(x,...) \leftarrow \phi\right) \in \Sigma$ **do**
  for all match θ of ϕ over $\Delta^{[0,i]}$ with $\phi\theta \cap \Delta^i \neq \emptyset$ **do**
    if $\Delta^{[0,i]} \not\models \left(\exists! id_H.id_H = H(x,y,...)\right)\theta$ **then**
      $\theta' = \theta \cup \{id_H \mapsto n\}$ where n is a fresh null value
      $\Delta^{i+1} = (\Delta^{i+1} \cup \left(id_H = H(x,...)\right)\theta') \setminus \Delta^{[0,i]}$
    end if
  end for
end for
$i = i + 1$

Example 2.2. Consider the following two $\mathcal{DL}^{\exists!}$ rules:

$$\begin{aligned} \exists! id_T.\ id_T = T(id_G) &\leftarrow id_G = G(x) \wedge x \neq A() \\ \exists! id_{T'}.\ id_{T'} = T(id_{G'}) &\leftarrow id_T = T(id_G) \\ &\quad \wedge id_G = G(id_{G'}) \wedge id_{G'} = G(x) \end{aligned}$$

Consider we start with $\mathcal{I}$ such that:

$$\begin{aligned} \Delta^0 = \{\ & id_A = A(), id_{G1} = G(id_A = A()) \\ & , id_{G2} = G\big(id_{G1} = G(id_A = A())\big)\} \end{aligned}$$

Notice that the database is referentially closed in the sense that every referenced sub-fact is included in Δ^0; we will more rigorously define this property shortly. Continuing the chase, we conclude:

$$\begin{aligned} \Delta^1 &= \{id_{T2} = T(id_{G2} = G(id_{G1} = G(id_A = A())))\} \\ \Delta^2 &= \{id_{T1} = T(id_{G1} = G(id_A = A()))\} \\ \Delta^3 &= \emptyset \end{aligned}$$

Like Datalog$^{\pm}$ and Datalog$^{\exists}$, the restricted chase for $\mathcal{DL}^{\exists!}$ is undecidable in general, evidenced by the fact that it is possible to build an interpreter for the λ calculus in $\mathcal{DL}_{\mathcal{S}}$ (Figure 3). However, the behavior of the restricted chase given in Algorithm 1 is simplified by the fact that facts are uniquely identified. This fact eliminates the challenge of unification, as facts are simply assigned a primary key associated with a Skolem term. With respect to chase termination, the programs for which Algorithm 1 does not terminate are precisely those which generate an unbounded number of *id*s. For any $\mathcal{DL}^{\exists!}$ program which produces facts bounded by any finite n, Algorithm 1 necessarily terminates; in § 3.1, we use this fact to establish the decidability of the Datalog subset of $\mathcal{DL}^{\exists!}$.

2.2 From $\mathcal{DL}^{\exists!}$ to $\mathcal{DL}_{\mathcal{S}}$

$$\begin{aligned} \langle Prog\rangle &::= \langle Rule\rangle^* \\ \langle Rule\rangle &::= R(\langle Subcl\rangle,\ldots) \leftarrow \langle Clause\rangle,\ldots \\ \langle Clause\rangle &::= id = R(\langle Subcl\rangle,\ldots) \\ \langle Subcl\rangle &::= R(\langle Subcl\rangle,\ldots) \mid \langle Var\rangle \mid \langle Lit\rangle \\ \langle Lit\rangle &::= \langle Number\rangle \mid \langle String\rangle \mid \ldots \end{aligned}$$

Figure 1: Syntax of $\mathcal{DL}_{\mathcal{S}}$: *R* is a relation name.

We now extend $\mathcal{DL}^{\exists!}$ to $\mathcal{DL}_{\mathcal{S}}$, syntactic sugar on top of $\mathcal{DL}^{\exists!}$ which allows syntactically-nested facts and patterns, such as $G(G(x))$. The syntax of $\mathcal{DL}_{\mathcal{S}}$ is shown in Figure 1. As in Datalog, a $\mathcal{DL}_{\mathcal{S}}$ program is a collection of Horn clauses. Each rule R contains a set of body clauses and a head clause, denoted by $Body(R)$ and $Head(R)$ respectively. $\mathcal{DL}_{\mathcal{S}}$ programs must also be well-scoped: variables appearing in a head clause must also be contained in the body; notice that $\mathcal{DL}_{\mathcal{S}}$ omits the existential quantification and assignment in the head, making the quantifier implicit. The definition of $\mathcal{DL}_{\mathcal{S}}$ is given via a syntax-directed translation given in Figure 2, which traverses rule bodies top-down to flatten their structure, decomposing nested clauses and flattening them to expose explicit identities. For each subclause, the *subcl* function is used to recursively flatten it to a set of flat clauses and a single value that uniquely identifies the original subclause without containing any structured subclauses.

$$\begin{aligned} &flatten : \langle Rule\rangle \rightarrow \langle Rule\rangle \\ &flatten(rule) \triangleq Head(rule) \leftarrow \bigcup_{cl \in Body(rule)} clause(cl) \\ &clause : \langle Clause\rangle \rightarrow \mathcal{P}(\langle Clause\rangle) \\ &clause(id = R(\ldots, scl_j)) \triangleq \{id = R(\ldots, scl'_i, \ldots)\} \cup \bigcup_i cset_i \\ &\quad \text{where } scl'_i, cset_i = subcl(scl_i) \\ &subcl : \langle Subcl\rangle \rightarrow \langle Subcl\rangle \times \mathcal{P}(\langle Clause\rangle) \\ &subcl(R(\ldots, scl_j)) \triangleq id, \{id = R(\ldots, scl'_i, \ldots)\} \cup \bigcup_i cset_i \\ &\quad \text{where } scl'_i, cset_i = subcl(scl_i) \text{ and } id \text{ is fresh} \\ &subcl(scl) \triangleq scl, \{\} \text{ where } scl \in \langle Var\rangle \cup \langle Lit\rangle \end{aligned}$$

Figure 2: Compiling $\mathcal{DL}_{\mathcal{S}}$ into $\mathcal{DL}^{\exists!}$

2.3 Fixed-point Semantics

The fixed-point semantics of a $\mathcal{DL}_{\mathcal{S}}$ program P is given via the least fixed point of an *immediate consequence* operator $IC_P : DB \rightarrow DB$. Intuitively, this immediate consequence operator derives all of the immediate implications of the set of rules in P. A database db is a set of facts ($db \in DB = \mathcal{P}(Fact)$). A fact is a clause without variables:

$$Fact ::= R(Val,\ldots) \qquad Val ::= R(Val,\ldots) \mid Lit$$

In Datalog, *Val*s are restricted to a finite set of atoms ($Val_{DL} ::= Lit$). To define IC_P, we first define the immediate consequence of a rule $IC_R : DB \rightarrow DB$, which supplements the provided database with all the facts that can be derived directly from the rule given the available facts in the database:

$$\begin{aligned} IC_R(db) \triangleq db \cup \bigcup \big\{ & subfact(Head(R)\big[\overrightarrow{v_i \backslash x_i}\big]) \mid \\ & \{\overrightarrow{x_i \rightarrow v_i}\} \subseteq (Var \times Val) \wedge Body(R)\big[\overrightarrow{v_i \backslash x_i}\big] \subseteq db \big\} \end{aligned}$$

The *subfact* function has the following definition:

$$\begin{aligned} subfact\big(R(item_1, \ldots, item_n)\big) &\triangleq \{R(item_1, \ldots, item_n)\} \\ &\quad \cup \textstyle\bigcup_{i \in 1\ldots n} subfact(item_i) \\ subfact(v)_{v \in Lit} &\triangleq \{\} \end{aligned}$$

The purpose of the *subfact* function is to ensure that all nested facts are included in the database, a property we call *subfact-closure*. This is also the semantic counterpart to *flatten* from Figure 2. The immediate consequence of a program is the union of the immediate consequence of each of its constituent rules, $IC_P(db) \triangleq$

$db \cup \bigcup_{R \in P} IC_R(db)$. Observe that IC_P is monotonic over the lattice of databases whose bottom element is the empty database. Therefore, if IC_P has any fixed points, it also has a least fixed point [94]. Iterating to this least fixed point directly gives us a naïve, incomputable fixed-point semantics for $\mathcal{DL}^{\exists!}$ programs. Unlike pure Datalog, existence of a finite fixed point is not guaranteed in $\mathcal{DL}^{\exists!}$ This is indeed a reflection of the fact that $\mathcal{DL}^{\exists!}$ is Turing-complete. The $\mathcal{DL}^{\exists!}$ programs whose immediate consequence operators have no finite fixed points are non-terminating. Results formalized in Isabelle/HOL proof assistant are marked with .

Lemma H1 . *The least fixed point of IC_P is subfact-closed.*

It is worth pointing out that the fixed point semantics of Datalog is similar, the only difference being that the *subfact* function is not required, as Datalog clauses do not contain subclauses.

2.4 Model-theoretic Semantics

The model-theoretic semantics of $\mathcal{DL}^{\exists!}$ closely follows the model theoretic semantics of Datalog, as presented in, e.g., [24]. The *Herbrand universe* of a $\mathcal{DL}^{\exists!}$ program is the set of all of the facts that can be constructed from the relation symbols appearing in the program. Because $\mathcal{DL}^{\exists!}$ facts can be nested, the Herbrand universe of any nontrivial $\mathcal{DL}^{\exists!}$ program is infinite. For example, $\mathbb{N}$ may be encoded in $\mathcal{DL}^{\exists!}$ using the zero-arity relation `Zero` and a unary relation `Succ`. The Herbrand universe produced by just these two relations, one zero arity and one unary, is inductively infinite.

A *Herbrand Interpretation* of a $\mathcal{DL}^{\exists!}$ program is any subset of its Herbrand universe that is subfact-closed. I.e., if I is a Herbrand Interpretation, then $I = \bigcup\{subfact(f) | \ f \in I\}$. For Datalog, the Herbrand Interpretation is defined similarly, with the difference that subfact-closure is not a requirement for Datalog, as Datalog facts do not contain subfacts.

Given a Herbrand Interpretation I of a $\mathcal{DL}^{\exists!}$ program P, and a rule R in P, we say that R is true in I ($I \models R$) iff for every substitution of variables in R with facts in I, if all the body clauses with those substitutions are in I, so is the head clause of R with the same substitutions of variables.

$$I \models R \text{ iff } \forall\{\overrightarrow{x_i \longrightarrow v_i}\} \ . \ Body(R)\left[\overrightarrow{v_i \backslash x_i}\right] \subseteq I \longrightarrow Head(R)\left[\overrightarrow{v_i \backslash x_i}\right] \in I$$

If every rule in P is true in I, then I is a *Herbrand model* for P. The denotation of P is the intersection of all Herbrand models of P. We define $\mathbf{M}(P)$ to be the set of all Herbrand models of P, and $D(P)$ to be P's denotation. Then, $D(P) \triangleq \bigcap_{I \in \mathbf{M}(P)} I$. Such an intersection is also a Herbrand model:

Lemma H2 . *The intersection of a set of Herbrand models is also a Herbrand model.*

Unlike Datalog, nontrivial $\mathcal{DL}^{\exists!}$ programs have Herbrand universes that are infinite. Thus, a $\mathcal{DL}^{\exists!}$ program may have only infinite Herbrand models. If a $\mathcal{DL}^{\exists!}$ program has no finite Herbrand models, its denotation is infinite and so no fixed-point may be finitely calculated using the fixed-point semantics. We now relate the operational semantics of $\mathcal{DL}^{\exists!}$ to its model-theoretic semantics: we elide a detailed proof, but refer the reader to our Isabelle implementation.

2.5 Equivalence

To show that the model-theoretic and fixed-point semantics of $\mathcal{DL}^{\exists!}$ compute the same Herbrand model, we need to show that the least fixed point of the immediate consequence operator is equal to the intersection of all the Herbrand models for any program.

Lemma H3 . *Herbrand models of a $\mathcal{DL}^{\exists!}$ program are fixed points of the immediate consequence operator.*

Lemma H4 . *Fixed points of the immediate consequence operator of a $\mathcal{DL}^{\exists!}$ program that are subfact-closed are Herbrand models of the program.*

By proving that the Herbrand models and subfact-closed fixed points of the immediate consequence operator are the same, we conclude that the least fixed point of the immediate consequence operator IC_P (a subfact-closed database) is equal to the intersection of all its Herbrand models.

Theorem H1 . *The model theoretic semantics and fixed point semantics of $\mathcal{DL}^{\exists!}$ are equivalent.*

3 APPLICATIONS OF FIRST-CLASS FACTS

We now ground our exploration of $\mathcal{DL}^{\exists!}$ in some familiar frameworks. To begin with, we observe that $\mathcal{DL}^{\exists!}$ strictly extends Datalog: any Datalog program is a terminating $\mathcal{DL}^{\exists!}$ program (and vice-versa, though with a potential complexity blowup). However, the true power of $\mathcal{DL}^{\exists!}$ lies in its ability to express queries wherein first-class facts are recursively (a) used as triggers and (b) generated by the computation. In effect, the combination of first-class rules and recursion enables ad-hoc polymorphic rules, which enable an expressive programming style similar to functional programming and natural deduction. We explore this power by a series of applications including provenance, algebraic data types, functional programming, structural abstract interpretation, and type systems.

3.1 Datalog in $\mathcal{DL}^{\exists!}$ and $\mathcal{DL}_{\mathcal{S}}$

Definition 3.1 (Datalog programs in $\mathcal{DL}^{\exists!}$ and $\mathcal{DL}_{\mathcal{S}}$). The Datalog subset of $\mathcal{DL}^{\exists!}$ is a subset wherein all occurrences of the binders for *id*s in the body are wildcards.

Theorem 3.2 (Datalog Programs Terminate). *If P is a Datalog program in $\mathcal{DL}^{\exists!}$, or $\mathcal{DL}_{\mathcal{S}}$, then the program terminates.*

Proof. For $\mathcal{DL}^{\exists!}$, termination follows directly from the fact that the restricted chase for Datalog programs without existentials necessarily has a finite number of iterations: because the bodies of rules ϕ in Algorithm 1 may not bind *id* columns (only wildcards), there is no ability to build a Skolem term of non-trivial height. Thus, as in Datalog, the restricted chase for $\mathcal{DL}^{\exists!}$ terminates. The termination for $\mathcal{DL}_{\mathcal{S}}$ follows similarly: because the body may not bind any *id*s in the variables x_i, the height of terms produce by *subfact* may not increase throughout iterated application of IC_R. □

3.2 Provenance in $\mathcal{DL}^{\exists!}$

Database provenance refers to the concept of tracking the origin of a database record. In Datalog and knowledge-graph reasoning, the coarsest and most popular form of provenance is lineage (why-provenance) which, for each tuple in the output (IDB), identifies

a set of contributing tuples from the input (EDB) [25, 31]. Typical implementations of lineage eagerly annotate each tuple with a tag during query computation to track its origin, and Datalog rules are rewritten to propagate the derivation order of these tags. After the query finishes, lineage is collected by computing the downward closure on the derivation graph, gathering all reachable leaf nodes to explain the tuple's origin. Such annotations are naturally isomorphic to the *id* values generated when evaluating existentially quantified queries in $\mathcal{DL}^{\exists!}$. Meanwhile, $\mathcal{DL}^{\exists!}$'s ability to directly construct nested facts naturally facilitates eager derivation computation. The derivation graph can be represented as a binary relation, *deriv*, which tracks all tuple *id*s from the body clauses of a Datalog rule to its head clause. For example, the derivation of the rule:

$$H(a, c) \leftarrow B_0(a, b) \wedge B_1(b, c)$$

can be a computed via:

$$\exists! id_3, id_4.\ id_3 = deriv(id_0, id) \wedge id_4 = deriv(id_1, id) \leftarrow id_0 = B_0(a, b) \wedge id_1 = B_1(b, c) \wedge id = H(a, c).$$

After constructing the derivation graph, the downward closure can be efficiently computed as a reachability query from the IDB tuple *id* to the EDB, identifying all contributing input tuples.

The eager approach stores complete provenance information for every tuple in the Datalog relation. Once the rule is computed, further analysis can be easily performed by querying the generated provenance relation. However, eagerly tracking provenance in datalog is data-intensive, with its time complexity proven to be NP [18]. In some specific applications, such as debugging, only the provenance of a few specific tuples is required. A cheaper but incomplete alternative is to compute the provenance in an lazy approach (also called on-demand approach). Instead of generating the full provenance during query evaluation, this method computes provenance only for the specific tuple that needs explanation or debugging. This involves extracting constraint formulas from the input and output databases and solving them using techniques such as SAT [19] or semi-ring solvers [36].

Although $\mathcal{DL}^{\exists!}$'s semantics eagerly annotates all tuples—naïvely yielding eagerly-materialized provenance—a hybrid approach can still be adopted by using a first-class fact to trigger lineage computation, similar to lineage tracking in the WHIPS system [30]. To lazily materialize the lineage, we may modify the previous rule:

$$\exists! id_5, id_6.\ id_5 = explain_t(id_0) \wedge id_6 = explain_t(id_1) \leftarrow id_0 = B_0(a, b) \wedge id_1 = B_1(b, c) \wedge id_3 = H(a, c) \wedge id_4 = explain_t(id).$$

Here, the *explain_t* relation constrains the search space. It initially stores the single tuple that needs to be explained, and during the computation, it is gradually expanded to include all tuples that cover the nodes in the derivation paths leading to the initial tuple. After *explain_t* has been propagated, the EDB relation IDs stored within it will be the why-provenance associated with the initial tuple needing explanation.

While why-provenance tracks the origin of each output tuple by annotating entire rows, where-provenance goes further by annotating individual columns [17]. This approach identifies the exact source locations (i.e., specific columns in the input data) that contribute to each value in the output, allowing for fine-grained tracing of data lineage at the attribute level.

In $\mathcal{DL}^{\exists!}$, a naïve way to capture where-provenance is to associated each concrete column value in a EDB relation with a tuple in *column* relation, whose id is isomorphic to the column annotation in definition of where-provenance. For example, the where-provenance $prov_H$ of relation H in query $H(a, c) \leftarrow B_0(a, b) \wedge B_1(b, c)$ can be materialized as:

$$\exists! id_H.id_H = prov_H(id_{a0}, id_{c1}) \leftarrow id_0 = B_0(a, b) \wedge id_1 = B_1(b, c) \quad id_{a0} = column(id_0, 0, a) \wedge id_{b0} = column(id_0, 1, b) \wedge id_{b1} = column(id_1, 0, b) \wedge id_{c1} = column(id_1, 1, c)$$

Here, *column* is a ternary relation, where $column(i, k, v)$ indicate the k-th column value of tuple associate with identity i is v.

3.3 Algebraic data types

Some Datalogs have a dedicated system for algebraic data types (ADTs) supporting records, unions, and tagged variants. For example, Soufflé offers a useful ADT system with heap-allocated structured values. In Soufflé, a heap-allocated structure is denoted with a $ sign; so, the following example is a Soufflé rule that says if a lambda exists as a subexpression of any parent expression (_), then its body eb is a subexpression of the lambda abstraction:

```
subexpr($lam(x, eb), eb) :- subexpr(_, $lam(x, eb)).
```

Unfortunately, ADTs cannot trigger rule evaluation and are not indexed—which has a performance impact we explain further in §5. For this reason, Soufflé requires we assert that the lambda is a subexpression of any expression since the lambda-syntax value must be referenced from some fact before it can be manipulated by a rule. Soufflé implements this rule as a scan of the **subexpr** relation, extraction of its second-column value, a lookup of that value's second field, a guard to check that it is a lambda, and finally, insertion of the lambda paired with its body-reference eb in the **subexpr** relation. By contrast, typical Datalog evaluation relies on relational joins, which do not normally require any guard but are implemented using an efficient B-tree lookup to determine the exact relevant tuples for the join. Following is an example that builds a transitive **within** relation from this **subexpr** relation. It is implemented in Soufflé using a relational union, a scan and insert, for the first rule, and a binary join (fused with a projection of the shared column) for the second rule. Operationally, this join would scan the outer relation **within**, lookup only those ec values that are paired with the relevant ei in **subexpr** using its tree-based index, and then insert each unique e,ec tuple into **within**. Here (Soufflé code), **within** is computed as the transitive closure of **subexpr**:

```
within(e, ec) :- subexpr(e, ec).
within(e, ec) :- within(e, ei), subexpr(ei, ec).
```

In Soufflé, since ADTs are heap allocated and not indexed the way (top-level) facts are, these values cannot be efficiently selected for in the course of a relational join operation. Because Soufflé cannot easily index its ADTs, a scan-and-filter approach is used. Consider a rule that computes a relation **shadows** pairing each lambda with each other lambda that defines a shadowing variable.

```
shadows($lam(x, eb0), $lam(x, eb1)) :-
     subexpr(_, $lam(x, eb0)), within(eb0, e),
     subexpr(e, $lam(x, eb1)).
```

Operationally, this is implemented as a three-way join between **subexpr**, **within**, and **subexpr** (which takes advantage of indexes for these relations), introspecting on the outer lambda to determine its body value (**eb0**) during this join, followed by introspection on the inner lambda to determine whether both x values match, before inserting relevant pairs of lambdas into **shadows**. This implicitly materializes the Cartesian product of nested lambda values encountered during the join (in time, not space) before filtering to only insert those with matching formal parameters. Because $lam values are scattered in memory and only referenced from top-level relations and are not collectively indexed, there is no straightforward way to efficiently select just those $lam values that are relevant to a particular x as we could when selecting **subexpr** values with a particular e. The code generated by Soufflé for a similar case, from a program analysis, is shown in Figure 7.

3.4 Functional Programming

With first-class facts that *can trigger* rule evaluation, and that are indexed automatically and amenable to efficient joins, natural idioms emerge for performing efficient functional programming within Datalog, that will benefit from data-parallel deduction as detailed in Section 4. First-class facts permit Reynolds' *defunctionalization* [84], a classic transformation from higher-order functions to first-order functions with ad hoc polymorphism over structured values. Note that first-class facts gives us precisely the features we need to implement this: structured data that can trigger rule evaluation via (naturally polymorphic) joins on their identity!

To illustrate this, consider the λ-calculus interpreter presented in Figure 3. The upper left shows the traditional big-step (natural-deduction style) rules of inference, that define relation ($\Downarrow$), for evaluating variable references, lambda abstractions, and lambda applications. The right side shows a corresponding interpreter in $\mathcal{DL}_\mathcal{S}$. The bottom left gives rules for defunctionalized higher-order environments built using a chain of ($\mapsto$) facts. E.g., the first-class map $[0 \mapsto 1, 2 \mapsto 3]$ can be constructed as $\mapsto(\mapsto(\bot(), 0, 1), 2, 3)$, using these rules, and then accessed efficiently via *lookup*.

A 3-way join between this *lookup* relation, a syntax-encoding *ref* relation, and an *eval* relation, implements variable reference in our interpreter. The *eval* relation encodes an input e, ρ pair so the interpreter can be demand driven; the ($\Downarrow$) relation associates an *eval* input with its output, the value it denotes. The second rule joins the *eval* and (λ) relations, and builds closures, *clo* facts, that are associated with the *eval* fact in ($\Downarrow$). The final App rule is encoded as four $\mathcal{DL}_\mathcal{S}$ rules because it makes three recursive uses of the interpreter. Our implementation of $\mathcal{DL}_\mathcal{S}$ as a full language, Slog (discussed in Section 4) provides syntactic sugar that will perform this transformation of a single rule into multiple rules for the user automatically, but in this paper we present it desugared to focus on the key principles at play. The rule shown with a conjunction in the head is really two $\mathcal{DL}_\mathcal{S}$ rules that infer both subexpressions of an application must be evaluated before the application itself. The bottom rule deduces that once these subexpressions have been evaluated, the body of the function being applied must be evaluated under an extended environment that binds the formal parameter

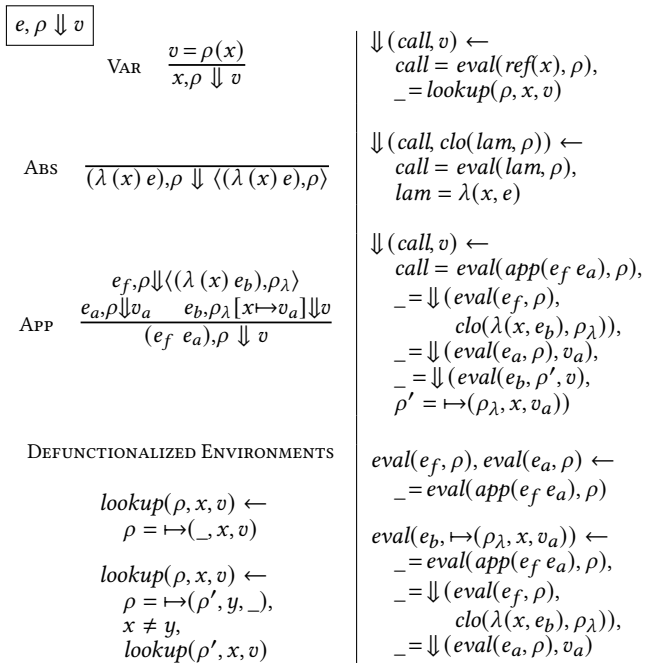

Figure 3: Lambda-calculus interpreter ($\Downarrow$) in $\mathcal{DL}_\mathcal{S}$.

(x) to the argument value (v_a) using a ($\mapsto$) fact. Finally, the top rule (shown parallel to App on the left) puts this all together and deduces that the application expression's value is the same as that of the function body under the appropriate environment.

IncA, the other recent approach to encoding functional programming in Datalog [77, 78] requires static monomorphization as a compilation approach because it is not able to directly leverage first-class facts to perform proper defunctionalization. Our extension, $\mathcal{DL}^{\exists!}$, grants us an efficient implementation approach that may be extended to a scalable system for performing data-parallel functional programming in Datalog. Sections 4 and 5 present further details of our practical system and evaluate it on applications that leverage the functional techniques we've discussed.

3.5 Structural Abstract Interpretation

A principal application motivating recent developments in Datalog has been high-performance declarative simulations of complex systems, especially software. Static program analyses aim to model software soundly based on the program text alone, with sufficient precision to prove meaningful correctness, security, observational equivalence, information flow, and other properties. Over the past decade or so, a line of work has gone into systematic and tunable approaches to applying *abstract interpretation* (AI) [27–29] to structural operational semantics [50, 82] and natural semantics [55] via a structural abstraction that is as straightforward as possible. The *abstracting abstract machines* (AAM) methodology [67, 100] prescribes a particular systematic application of abstract interpretation on abstract-machine operational semantics like those we built in §3.4—this approach is a natural one for traditional Datalog-based analyses as it requires a flat, small-step semantics with a particular

```
// Eval states
ret(v,k) :- eval($Ref(x),env,k,_),
    env_map(x,env,a), store(a,v).
ret($Clo($Lam(x,body),env),k) :-
    eval($Lam(x,body),env,k,_).
eval(ef,env,$ArK(ea,env,$App(ef,ea),c,k),c) :-
    eval($App(ef,ea),env,k,c).
// Ret states
eval(ea,env,$FnK(vf,call,c,k),c) :-
    ret(vf,$ArK(ea,env,call,c,k)).
apply(call,vf,va,k,c) :- ret(va,$FnK(vf,call,c,k)).
ret(v,k) :- ret(v,$KAddr(e,env)),kont_map($KAddr(e,env),k).
program_ret(v) :- ret(v, $Halt()).
// Apply states
eval(body,$UpdateEnv(x,
    $Address(x,[call,[hist0,[hist1,nil]]]),env),
    $KAddr(body,$UpdateEnv(x,
        $Address(x,[call,[hist0,[hist1,nil]]]),env)),
                        [call,[hist0,[hist1,nil]]]),
kont_map($KAddr(body,
    $UpdateEnv(x,$Address(x,
        [call,[hist0,[hist1,nil]]]),env)),k),
store($Address(x,[call,[hist0,[hist1,nil]]]),va),
env_update(env,$UpdateEnv(x,
    $Address(x,[call,[hist0,[hist1,nil]]]),env)) :-
    apply(call,$Clo($Lam(x,body),env),va,k,[hist0,[hist1,_]]).
// Propagate free vars
env_map(x,env1,a) :- env_update(env,env1),
     env1 = $UpdateEnv(x,a,env).
env_map(x,env1,a) :- env_update(env,env1),
     env_map(x,env,a).
```

Figure 4: A global-store m-CFA—evaluated in Table 2.

abstraction of the stack (once explicitly modeled in the semantics). Some AAM-based approaches have encapsulated the abstraction in a tunable monadic interpreter [89]. A related trend toward *abstracting definitional interpreters* (ADI) [13, 32] follows a similar approach, except applies AI directly to naturally recursive definitional interpreters, sometimes using a monad transformer stack for tunable abstraction and a partial-evaluation-based approach to achieve performance in the implementation [3, 102, 103].

The literature on the AAM approach shows that a *store-passing* transformation (as might also be used to model, *e.g.*, direct mutation) can handle indirect recursion in the *abstract* domains (*e.g.*, binding environments contain closures which contain environments) through a set of references that are finitized under AI [67]. This forms a key preparatory transformation enabling a straightforward homomorphic structural abstraction to follow. For any abstract-machine component referenced through the store, the *allocation* of its *abstract addresses* becomes an expressive proxy for tuning the polyvariance (e.g., context sensitivity, flow sensitivity) of those analysis components, and one with the notable property that soundness can be guaranteed for all tunings of abstract allocation [43, 68]. This is likewise the natural approach for continuation-passing abstract machines that store-allocate continuations at function calls [100], and is a key enabling technique as AAMs require some way of finitizing otherwise-unbounded recursive evaluation. A particular reflective choice of address for continuations (the function entry point, in the analysis, including its abstract environment, after parameters are bound) can guarantee the analysis is equivalent to a pushdown system with an unbounded stack [46]. A wide variety of heavyweight verification tasks can be built atop this foundation, including higher-order pointer and shape analysis [41, 42, 69] and abstract symbolic execution [75, 96].

We implement a core AAM-based analysis that permits an apples-to-apples comparison between our system Slog that implements

$\boxed{\Gamma \vdash e : \tau}$

$$\text{T-Var}\quad \frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \left|\begin{array}{l} id_0 = ck(ref(x), \Gamma), \\ _ = lookup(\Gamma, x, T) \\ \rightarrow type(id_0, T) \end{array}\right.$$

$$\text{T-Abs}\quad \frac{\Gamma, x : T_1 \vdash e : T_2}{(\lambda\,(x : T_1)\; e) : T_1 \rightarrow T_2} \qquad \left|\begin{array}{l} id_0 = ck(\lambda(x, T_1, e), \Gamma), \\ _ \;\; = type(ck(e, \rho'), T_2) \\ \rho' \;= \mapsto(\Gamma, x, T_1) \\ \rightarrow type(id_0, arrow(T_1, T_2)) \end{array}\right.$$

$$\text{T-App}\quad \frac{\Gamma \vdash e_0 : T_0 \rightarrow T_1 \qquad e_1 : T_0}{\Gamma \vdash (e_0\;\; e_1) : T_1} \qquad \left|\begin{array}{l} id_0 = ck(app(e_0, e_1), \Gamma), \\ _ = type(ck(e_0, \Gamma), arrow(T_0, T_1)), \\ _ = type(ck(e_1, \Gamma), T_0), \\ \rightarrow type(id_0, T_1) \end{array}\right.$$

Figure 5: STLC type rules (left); equivalent $\mathcal{DL}_S$ (right).

$\mathcal{DL}^{\exists!}$ (see §4), and Soufflé (see §3.3). It is derived from a stack-passing interpreter—a call-by-value Krivine's machine [58]—with a store factored out. Figure 4 shows our small-step control-flow analysis (CFA) in Soufflé with a global store and a tunable instrumentation. The original k-CFA [90] used true higher-order environments, unlike equivalent analyses written for object-oriented languages which used flat objects. Our analysis in Figure 4 is implemented as the corresponding CFA for functional languages, called m-CFA [70], and is used in our evaluation to compare first-class fact vs ADT handling in Slog vs Soufflé (see §5.2).

3.6 Type Systems

Structural type theories provide a formal framework for ensuring that programs are free from runtime type errors [80]. Such systems grew out of a rich history of constructive logic (e.g., Per Martin-Löf's intuitionistic type theory [65]), typically specifying typing rules via natural deduction, and form the basis of type checking and inference systems in myriad modern programming languages, such as OCaml and Haskell. Implementing type checkers for structural type systems involves iteratively traversing a program's abstract-syntax tree and attempt to assemble a valid type derivation which adheres to the typing judgement for the given language.

Traditionally, structural type systems have been implemented in functional programming languages, using recursion and pattern matching. However, Datalog has attracted recent attention in implementing type systems due to its logic-defined nature, along with the potential for automatic incrementalization and high-performance compilation [76, 78]. These efforts are challenged by Datalog's semantic restrictions: type systems often involve traversing tree-shaped abstract syntax trees and assembling tree-shaped typing derivations. As such, Datalog-based type systems require tedious refactoring of the type system, carefully compiling type checking algorithms into rules which respect Datalog's restrictions via techniques such as monomorphization, ultimately yielding implementations which are far removed from the original typing rules.

We have found that $\mathcal{DL}^{\exists!}$ has proven a promising implementation candidate for structural type systems, and have used $\mathcal{DL}_S$ to implement numerous type checkers for a range of systems including first-order dependent types [81] and intuitionistic type theory [65]. As an example, Figure 5 shows a transliteration of the

Simply-Typed Lambda Calculus (STLC) into $\mathcal{DL}_{\mathcal{S}}$, following a textbook presentation [80]. The left-hand side of the figure displays the three type rules for STLC. The first says that if x has type T in typing environment Γ, then x can be checked to the type T in Γ. This is mirrored by the $\mathcal{DL}_{\mathcal{S}}$ rule on the right, which uses the *ck* fact to trigger lookup of x (using the defunctionalized environments from the bottom of Figure 3) and materialization of the *type* judgement. The second rule triggers when a lambda is checked, subsequently extending the environment and checking the type of the body, finally assembling an function (*arrow*) type. Last, T-App checks an application, ensuring that the function's input (T_0) matches with e_1's type. In STLC, the simple nature of types means that $\mathcal{DL}^{\exists!}$'s structural equality suffices for type comparison' More complex type systems equate types, would also implementing a function to (*e.g.*,) canonicalize types. Last, we omit rules to trigger ck for subordinate expressions for brevity; our implementation uses syntactic sugar to enable writing a single rule for each of these cases.

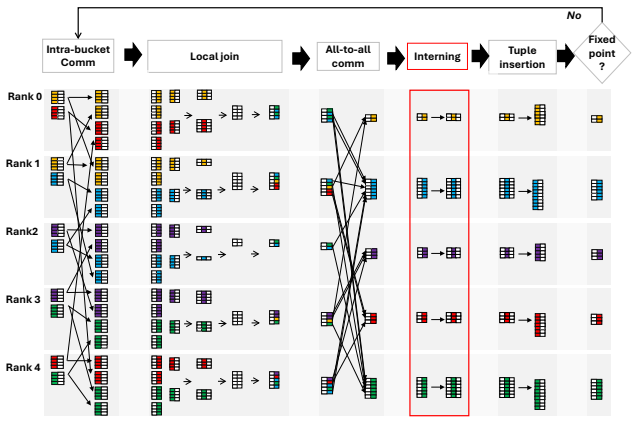

Figure 6: An illustration of the main phases of our parallel RA backend.

4 SLOG: IMPLEMENTING $\mathcal{DL}_{\mathcal{S}}$

In this section, we describe the implementation of $\mathcal{DL}_{\mathcal{S}}$ as a practical language. *Structured Datalog* (Slog) represents a language design and implementation methodology based on $\mathcal{DL}^{\exists!}$, Datalog extended with the principle that all facts are also first-class structured data and every structured datum is a first-class fact. We have implemented Slog in a combination of Racket (the compiler, roughly 10,600 new lines), C++ (the runtime system and parallel RA backend; roughly 8,500 new lines), Python (a REPL and daemon; roughly 2,500 new lines), and in Slog (60 lines for language feature support). In Slog, fact identities are implemented as 64-bit primary keys implemented via a distributed and parallel autoinc.

Our Racket-based compiler translates Slog source code to C++ that links against our parallel relational-algebra backend. Our compiler is structured in the nanopass style [57], composed of a larger number of small passes which consume and produce various forms of increasingly-low-level intermediate representations (IRs). After parsing, an organize pass performs various simplifications, such as canonicalizing the direction of implication (`-->`) and splitting disjunctive body clauses and conjunctive head clauses into multiple rules. This pass also eliminates various syntactic niceties of the language. Subsequent passes perform join planning, then index generation: computing a minimal necessary set of shared indices, ensuring there is always an interning index that models the relation as a mapping from fact-tuple values to an *unsigned 64-bit integer encoding a unique intern id for that fact.* The compiler then performs Strongly Connected Component (SCC) computation with Tarjan's algorithm [93] and stratifies grounded fact-negation operations and aggregation operations. Then, the compiler performs incrementalization, or the so-called semi-naïve transform in which rules are triggered by the delta of a relation accessed as a body clause—reducing the substantial recomputation of so-called naïve fixpoint evaluation. Finally, the compiler writes out a set of optimized and pass-fused relational algebra operations (e.g., join followed by projection) in C++ .

4.1 MPI-based Data-parallel Implementation

We extend an MPI-based Datalog back-end from our prior work on data-parallel relational algebra [44, 45, 62], to demonstrate that the key extension of Datalog we propose is naturally data-parallel, fact identity being derived from reads over indices in precisely the same data-parallel manner as rule evaluation in Datalog.

Our parallel relational-algebra (RA) backend supports fixed-point iterations and is designed for large-scale multi-node HPC clusters. Based on the bulk-synchronous-processing protocol and built using the `MPI-everywhere` model [40, 105], the parallel RA framework addresses the problem of partitioning and balancing workloads across processes by using a two-layered distributed hash table [44]. In order to materialize newly generated facts within each iteration, and thus facilitate *iterated* RA (in a fixed-point loop), an all-to-all data exchange phase is used at every iteration.

Figure 6 shows a schematic diagram of all the phases of our parallel relational algebra, including a new *interning* phase, in the context of Slog's fixed-point loop. There are five primary phases of our system: intra-bucket communication, local RA computation, all-to-all data exchange, interning, and materialization in appropriate indices. Relations are partitioned across processes using a modified double-hashing approach [99]. This involves partitioning relations by a hash of their join-column values so that they can be efficiently distributed to participating processes. The main insight behind this approach is that for each tuple in the outer relation, all relevant tuples in the inner relation must be hashed to the same bucket, stored on the same MPI process or node, permitting joins to be performed locally on each process. To handle any possible key-skew (imbalance across keys) in the relations themselves, we also hash the non-join columns to map the tuple to a sub-bucket. Each unique bucket/sub-bucket pair is then mapped to an MPI process. To distribute subbuckets to managing processes, we use round-robin mapping and have a mechanism to dynamically refine buckets that accumulate large numbers of tuples into a greater number of sub-buckets.

Intra-bucket Comm. and Local Join. Our double-hashing approach to manage key-skew necessitates an intra-bucket communication phase to co-locate matching tuples before the join: all sub-buckets for the outer relation, R_Δ, are transmitted to all other sub-buckets (in the same bucket) temporarily for the join. During the local computation phase, RA kernels (comprised of fused join, selection, projection, and union) are executed in parallel across all processes.

All-to-all Comm. Output tuples generated from local joins may each belong to an arbitrary bucket in the output relation, so a non-uniform all-to-all communication phase shuffles the output to their managing processes (preparing them for any subsequent iterations). Materializing a tuple in an output relation involves hashing on its join and non-join columns to find its bucket and sub-bucket (respectively), and then transmitting it to the process that maintains that bucket/sub-bucket. The overall scalability of the RA backend relies on the scalability of the all-to-all inter-process data exchange phase, but all-to-all is notoriously difficult to scale [60, 86, 95]—largely because of the *quadratic* nature of its workload. We address this scaling issue by adopting recent advancements [37] that optimize non-uniform all-to-all communication by extending the log-time Bruck algorithm [15, 95, 97] for non-uniform all-to-all workloads.

Deduplication, Interning, and Insertion. Once new tuples are received, we perform the interning phase required to assign unique first-class facts their unique identity. This first checks if the received fact was already discovered; if not, then a new 64-bit intern id is created and associated with the fact. This is done by reserving the first 16 bits of the 64-bit ID for the relation ID, the next 16 bits for the bucket ID, and the remaining 32 bits for a unique fact ID—generated on a per-process basis via simple bump-pointer allocation. Critical to our approach, when facts are generated, they are sent to the canonical index, so that fact-ID generation may happen in parallel on each bucket, as each bucket performs local insertion and deduplication into the canonical (master) index, without any additional communication or synchronization required. Each fact is then inserted into R_{new}, and following the semi-naïve evaluation approach, these newly generated facts form the input R_Δ for the following iteration of the fixed-point loop. This process continues until a fixed point is reached.

5 APPLICATIONS AND EVALUATION

In this section, we elaborate upon several applications of Slog; we directly evaluate the performance of Slog against several state-of-the-art tools in the context of provenance and program analysis.

5.1 Eager Why-Provenance (Lineage)

In prior literature, why-provenance has been deemed useful but eagerly computing it is considered too expensive because it requires storing all possible derivations of a tuple. By contrast, the on-demand approach calculates possible derivations for user-specified tuples after the execution. For this reason, existing Datalog engines compute why-provenance on-demand in a top-down fashion, thwarting effective data parallelism. Slog's parallel capabilities prompt us to again investigate eager provenance evaluation.

We explore this by comparing the running time for eager lineage calculation using Slog and three other Datalog engines: VLog, Nemo, and RDFox [51, 72, 98]. For VLog, we use its Java version, Rulewerk, which supports lineage computation via an existential operator. Nemo is another engine, similar to VLog, but implemented in Rust. RDFox is a high-performance commercial Datalog engine that supports lineage computation via a built-in Skolem function. These experiments were done on a machine with a 3.0Ghz 12-core AMD 5945WX (Zen3 Chagall) with 128GB of memory. We ran each experiment five times and report the mean; results are shown in Table 1. We provide single-thread performance comparisons for all engines, and for parallel RDFox and Slog, we also report the running times when all cores on our test platform are utilized.

Table 1: Running Time (s): Eager why-provenance vs. Nemo, Rulewerk (single threaded), RDFox; timeout (🕓) of one hour.

Query	Dataset	Nemo (Rust)	Vlog (Java)	RDFox (threads) 1	RDFox (threads) 12	Slog (threads) 1	Slog (threads) 12
Galen	15	1.63	7.38	0.20	0.14	0.68	**0.13**
	25	2.35	17.7	0.33	0.19	0.92	**0.17**
	50	34.5	415	2.30	**1.98**	12.0	2.19
CSDA	httpd	72.2	152	57.9	40.7	127	**22**
	linux	548	🕓	315	230	6,237	**116**
	postgres	234	🕓	138	95.2	320	**62.6**
Andersen	10,000	1.38	3.94	1.08	0.61	1.21	**0.19**
	50,000	7.70	22.6	6.43	3.04	7.84	**0.94**
	100,000	16.5	42.3	13.7	6.78	21.7	**2.28**
	500,000	119	386	78.8	30.4	144	**14.6**
TC	SF.cedge	439	296	443	291	628	**183**
	wiki-vote	235	1372	391	337	485	**91**
	Gnutella04	376	🕓	441	321	725	**124**

The first column of the table shows four selected Datalog queries. Galen is a Datalog version of the EL ontology [56] using the ELK calculus. The three datasets used in this query are ontologies found in the Oxford Library. CSDA is a context-sensitive null-pointer analysis. The three datasets here are collected from real-world applications by the authors of a static analysis tool called Graspan [101]. Andersen is a directed (Andersen-style) points-to flow analysis—the datasets for this query are extracted from synthesized program control-flow graphs with edge counts ranging from 10k–500k. Lastly, TC refers to a transitive closure query running on three real-world graphs from the SuiteSparse collection [33].

Columns 3, 4, 5, and 8 compare the single-core performance of all four engines. Overall, Slog demonstrates similar single-core performance to the Rust-based Datalog engine Nemo but is slower than RDFox. In the Galen query tests, Slog consistently outperforms Nemo, though it is slower in other benchmarks. The slowest performance for Slog is observed with the CSDA postgres dataset, where it takes 1.34× longer than Nemo. On average, Slog is 2× slower than RDFox in single-core performance. Despite this, Slog, along with Nemo and RDFox, consistently outperforms VLog across all test cases. The slowdown of single-threaded Slog in the CSDA and Andersen queries, which involve higher memory usage than the Galen queries, is likely due to the overhead of maintaining MPI buffers, necessary for multi-core execution. However, with 12

threads, these parallel MPI facilities become worthy, making Slog the fastest engine overall when fully utilizing multi-core.

Columns 6 and 8 show the running times of RDFox and Slog with 12 threads. Leveraging data-parallelism, both engines significantly outperform the single-thread engines in all test cases when running with 12 threads. At full utilization of 12 cores, Slog surpasses RDFox in performance when compared at the same core counts. Compared to single-process Slog, multi-process execution provides an average 6× speedup, while 12-thread execution only yields a 1.5× improvement. This indicates that Slog has better scalability and still has potential for further scaling with higher core counts, whereas RDFox's scalability appears to saturate at 12 cores.

We also benchmarked a tool that constructs on-demand why-provenance using a combination of Answer Set Programming (ASP) and SAT solvers [19]. We attempted to run it in parallel to demand the why-provenance of all tuples in the output to simulate eager provenance; we found that the SAT solver sometimes gets stuck on lineage for certain tuples, and was not comparable with the other tools due to the inability to reuse work.

5.2 Evaluating CFA: Slog facts vs. Soufflé ADTs

We have implemented the control-flow analysis from Figure 4 in both Soufflé and Slog, and have compared their runtimes at both 8 and 64 processes on a large cloud server (with 64 physical cores) on various large synthetic benchmarks; we evaluate both k-CFA (exponential) and m-CFA (polynomial), two forms of call sensitivity. We report our results in Table 2. Each of six distinct analysis choices is shown along the left side of the table. Along rows of the table, we show experiments for a specific combination of analysis, precision, and term size. We detail the total number of iterations taken by the Slog backend, along with control-flow points, store size, and runtime at both 8 and 64 threads for Slog and Soufflé. Times are reported in minutes / seconds form; several runs of Soufflé took under 1 second (which we mark with <0:01); ⏲ indicates timeout (over four hours).

Inspecting our results, we observed several broad trends. First, as problem size increases, Slog's runtime grows less-rapidly than Soufflé's. This point may be observed by inspecting runtimes for a specific set of experiments. For example, 10-m-CFA with term size 200 took Slog 26 seconds, while Soufflé's run took 56 seconds. Doubling the term size to 400 takes 104 seconds in Slog, but 398 seconds in Soufflé—a slowdown of 4× in Slog, compared to a slowdown of 7× in Soufflé. A similar trend happens in many other experiments, e.g., 15 minutes to over three hours for Soufflé (13× slowdown) vs. 2 to 4 seconds (2× slowdown) in Slog's runtime on 5-k-CFA. Inspecting the output of Soufflé's compiled C++ code for each experiment helped us identify the source of the slowdown. For example, the rule for returning a value to a continuation address `$KAddr(e,env)`, in Figure 4, must join a return state using this address with an entry in the continuation store for this address.

Consider the rule and its compiled C++ code shown in Figure 7. Note that it uses two nested for loops to iterate over the entire **`ret`** relation, then iterate over the entire **`kont_map`** relation, and finally check if there is a match, for every combination. In this way, Soufflé's lack of indices for structured values leaves it no other choice but to materialize (in time, not space) the entire Cartesian

Table 2: Control-Flow Analysis: Slog vs. Soufflé (ADTs)

	Size	Iters	Cf. Pts	Sto. Sz.	8 Processes		64 Processes	
					Slog	Soufflé	Slog	Soufflé
3-k-CFA	8	1,193	98.1k	23.4k	**0:01**	1:07	0:02	00:15
	9	1,312	371.0k	79.9k	**0:02**	14:47	0:03	02:56
	10	1,431	1.44M	291.3k	0:06	⏲	**0:05**	45:49
	11	1,550	5.68M	1.11M	0:27	⏲	**0:16**	⏲
	12	1,669	22.5M	4.32M	2:14	⏲	**1:07**	⏲
	13	1,788	89.8M	17.0M	12:17	⏲	**5:08**	⏲
4-k-CFA	9	1,363	311.8k	65.4k	**0:01**	14:38	0:03	02:08
	10	1,482	1.20M	229.6k	**0:05**	⏲	0:05	40:30
	11	1,601	4.69M	853.5k	0:20	⏲	**0:13**	⏲
	12	1,720	18.6M	3.28M	1:40	⏲	**0:53**	⏲
	13	1,839	73.8M	12.9M	8:44	⏲	**3:58**	⏲
	14	1,958	294.4M	50.9M	1:00:53	⏲	**35:46**	⏲
5-k-CFA	9	1,429	203.7k	50.7k	**0:02**	05:30	0:03	1:15
	10	1,548	756.9k	167.3k	**0:04**	65:20	0:04	15:08
	11	1,667	2.91M	597.5k	0:13	⏲	**0:08**	3:16:06
	12	1,786	11.4M	2.25M	0:56	⏲	**0:27**	⏲
	13	1,905	45.2M	8.69M	4:38	⏲	**2:00**	⏲
	14	2,024	179.9M	34.2M	25:14	⏲	**9:58**	⏲
10-m-CFA	50	6,120	21.0k	656.8k	0:02	0:02	00:10	**0:01**
	100	11,670	42.9k	2.78M	0:07	0:09	00:20	**0:04**
	200	22,770	86.6k	11.4M	0:26	0:56	00:42	**0:23**
	400	44,970	174.0k	46.4M	1:44	6:26	**01:38**	1:56
	800	89,370	348.8k	187.0M	7:35	45:22	**04:21**	9:33
	1600	178,170	698.4k	750.1M	32:56	⏲	**14:36**	1:02:35
12-m-CFA	25	3,559	17.1k	385.3k	0:01	**<0:01**	0:06	<0:01
	50	6,434	36.3k	1.89M	0:04	**0:03**	0:11	0:03
	100	12,184	74.6k	8.28M	0:16	00:24	0:23	**0:10**
	200	23,684	151.3k	34.7M	01:10	02:37	**0:53**	0:55
	400	46,684	304.7k	141.9M	05:04	18:39	**2:23**	4:12
	800	92,684	611.5k	573.8M	22:46	2:38:22	**7:28**	24:58
15-m-CFA	12	2,211	14.5k	136.7k	0:01	**<0:01**	0:04	<0:01
	24	3,591	35.9k	1.44M	0:03	0:02	0:06	**0:01**
	48	6,351	78.6k	8.29M	0:14	0:15	0:14	**0:07**
	96	11,871	164.2k	38.9M	01:08	01:41	**0:36**	**0:36**
	192	22,911	335.4k	168.0M	05:15	12:10	**1:49**	2:51
	384	44,991	678k	697M	24:10	1:32:35	**6:45**	16:32

product as it checks for matches—rather than efficiently selecting tuples via an index as would be typical for top-level relations.

For a fixed problem size, we found that Soufflé and Slog both scaled fairly well. Soufflé consistently performed well on small input sizes; additional processes did not incur slowdowns, and Soufflé's efficiency was generally reasonable (roughly 50%) when algorithmic scalability did not incur slowdowns. For example, in 3-k-CFA (n=8), Soufflé took 67 seconds at 8 processes, and 15 seconds at 64 processes. Slog's parallelism doesn't outweigh communication overhead on smaller problems, particularly on problems

```
// ret(av, k) :- ret(av, $KAddr(e, env)), kont_map($KAddr(e, env), k).
//        env0[0] ---^  env2[0]-^  ^--- env2[1]       env3[1] -----^
if(!(rel_13_delta_ret->empty()) && !(rel_18_kont_map->empty())) {
  for(const auto& env0 : *rel_13_delta_ret) {
    RamDomain const ref = env0[1];
    if (ref == 0) continue;
    const RamDomain *env1 = recordTable.unpack(ref,2);
    {
      if( (ramBitCast<RamDomain>(env1[0]) == ramBitCast<RamDomain>(RamSigned(3)))) {
        RamDomain const ref = env1[1];
        if (ref == 0) continue;
        const RamDomain *env2 = recordTable.unpack(ref,2);
        {
          for(const auto& env3 : *rel_18_kont_map) {
            // On this line we've ommitted bitcasts and Tuple ctors:
            if(!(rel_19_delta_kont_map->contains({{ {{env2[0],env2[1]}}, env3[1]}}))
                && !(rel_12_ret->contains({{env0[0], env3[1]}}))) {
              // Omitted: null checks and insertion of  env0[0], env3[1]  into ret
}}}}}}}}}
```

Figure 7: Example C++ code generated by Soufflé.

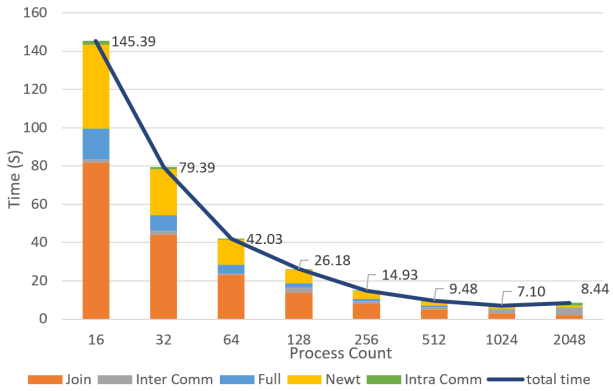

Figure 8: Scaling CSPA (of Linux) on Theta.

with high iteration count and low per-iteration work. As problem size increases, our Slog implementations show healthy scalability; efficiency grows as problem size grows (e.g., 24:10 to 6:45 on 15-*m*-CFA/384, 22:46 to 7:28 on 12-*m*-CFA/800).

We found that extracting optimal efficiency from Slog was facilitated by increasing per-iteration work and avoiding long sequences of sequential work with comparatively lower throughput; we believe this is because of the synchronization required to perform deduplication. As an example of this principle, our 10-*m*-CFA uses a flat `ctx` fact to represent the context; a previous version used a linked list 10 elements deep, however this design achieved poorer scaling efficiency due to these 10-iteration-long sequences of work necessary to extend the instrumentation at each call-site. In our experiments scaling efficiency improved as polyvariance increased; e.g., improving by 2× for 10-*m*-CFA, but 3.5× for 15-*m*-CFA. We believe this is because of the higher per-iteration work available.

5.3 Scaling CSPA on the Theta Supercomputer

The performance differences we have just observed are primarily due to the Slog's asymptotic benefits in compiling rules that manipulate algebraic data. It is also reasonable to ask whether Slog's data parallelism enables sufficient absolute performance in useful applications. To measure this, we transliterated (from [38]) a 10-rule vanilla-Datalog implementation of Context-Sensitive Points-to Analysis (CSPA) to run using Slog. We evaluated our implementation using several large-scale test programs from Graspan [101], including `httpd`, `postgres`, and `linux` (a Linux kernel without modules). We compiled our Slog version of CSPA to run using MPI and ran the resulting program at a variety of scales (from 16–2,048 processes) on the Theta Supercomputer [79] at the Argonne Leadership Computing Facility (ALCF) of the Argonne National Laboratory. Theta is a Cray machine with a peak performance of 11.69 petaflops. It is based on the second-generation Intel Xeon Phi processor and is made up of 281,088 compute cores. Theta has 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM, 10 PiB of storage, and a Dragonfly network topology.

Figure 8 shows the result of our strong-scaling runs for CSPA of Linux (using Graspan's data). Overall, we achieve near-perfect scalability up to around 512 processes where we observe a performance improvement commensurate to change in scale. After 512 processes, scalability starts to decline owing to an increase in data movement costs and workload starvation. This trend is typical of HPC applications, which typically have a range of processes for which we observe optimal scalability. While we could not run Soufflé on Theta, we observed roughly similar runtimes (around 150s for both Slog and Soufflé) for this experiment at 16 threads on a large Unified Memory Access (UMA) server. We see this result as a promising indicator of the potential for Slog to achieve high throughput, but it is worth noting that our data set (from Graspan) achieves context sensitivity via method cloning and thus provides a large amount of available parallel work. In our future work, we aim to use Slog to develop rich, context-sensitive analyses of fully-featured languages.

6 RELATED WORK AND LIMITATIONS

We now categorize several recent threads of related-work apropos our efforts. Along the way, we highlight limitations of our current implementation of Slog and discuss some plans for future work.

Datalog$^{\exists}$ and The Chase. Our $\mathcal{DL}^{\exists!}$ is a restriction of Datalog$^{\exists}$, a well-known extension of Datalog, which allows arbitrary existential existential quantification in the heads of rules [1]. Datalog$^{\exists}$ is particularly popular in the field of knowledge representation and reasoning (KRR), where it enables tractable query answering over ontologies [8]. As a result, many Datalog-based reasoners include this extension. However, existential queries are computationally expensive to compute and can also lead to undecidable queries. To address this, constrained versions of Datalog, such as those in the Datalog$^{\pm}$ family [20], are used to ensure decidability. These dialects employ various constraints, like those in the Graal reasoner [7] and the Shy system in the i-DLV [63].

Existentially quantified conjunctive queries create tuple generating dependencies (TGDs) between the head and body clauses of a Datalog$^{\exists}$ rule. To resolve these dependencies, variants of the well-known database algorithm, the Chase [64], are used. One complete version is the *oblivious chase* [12], which explores all possible choices for the existentially quantified meta variables. However, this approach leads to significant non-termination issues, making it impractical for mainstream Datalog$^{\exists}$ implementations. Instead, these engines often adopt algorithms with stronger termination guarantees, such as the *restricted chase* [21] and the *parsimonious chase* [63].

Semirings and provenance. The highly influential work of provenance semirings extends Datalog with K-relations, wherein every Datalog fact is labeled with a value from a semiring [47]. $\mathcal{DL}^{\exists!}$ is orthogonal to provenance semirings. In general, wile $\mathcal{DL}^{\exists!}$ does allow annotating tuples with arbitrary structured values, $\mathcal{DL}^{\exists!}$ does not equate tuples modulo the semiring laws, instead equating facts via their Skolem term. Thus, while $\mathcal{DL}^{\exists!}$ does allow encoding provenance semirings (*e.g.*, $\cdot(+(\ldots),\ldots)$), the need to materialize distinct Skolem terms (for identical semiring values) leads to encoding overhead. However, unlike provenance semirings, $\mathcal{DL}^{\exists!}$ does allow observing the tuple's identity: while provenance semirings forbid introspecting upon tuple tags in the Datalog program, $\mathcal{DL}^{\exists!}$ enables matching on structured facts to drive computation.

Aside from provenance semirings, there has been significant interest in the application of provenance for a variety of applications, including data warehousing [31], debugging Datalog [107], explainable AI [22, 35, 39], and program analysis [25]. Provenance has seen particular recent interest in explainable AI; our comparison systems Rulewerk and Nemo [34, 51] are representatives of this work, supporting why-provenance via an existential operator. Recent research leverages Answer Set Programming (ASP) and SAT solvers for why-provenance [19]. However, all the above approaches take an on-demand approach to computing why provenance, deeming eager materialization as unnecessary or overly expensive. By contrast, $\mathcal{DL}^{\exists!}$ focuses on ubiquitous, eager materialization of tree-shaped facts; our high-performance, data-parallel RA kernels (§ 4.1) enable representing provenance in a compact, distributed fashion.

Equality in Datalog. Equality-generating dependencies (EGDs) are an extension of Datalog that allow the equality operator (=) to appear in the head clauses of rules. This extension has become favored in Datalog due to its ability to facilitate new realms of application, such as financial data analysis [9] and program equality saturation [104]. One way to enable EGD reasoning is by storing Datalog relations in specialized union-find-like data structures called equality graphs (e-graphs). Of particular note is Egglog [106], which allows programming with equality sets backed by a lazy-building e-graph implementation called egg. Though techniques like lazy-building can be used to accelerate computation, full EGD chasing is still expensive. Thus, some Datalog engines, such as Vadalog, choose to use a stratified EGD semantics called *harmless EGD* [11], which avoids the values generated in EGD ruless from being further used to trigger the activation of other rules. We view Slog as complementary to Egglog, with similar but diverging goals and substantive differences: Slog does not support equality sets, though it is possible to materialize an equivalence relation (either lazily or eagerly) for bounded observations. Additionally, Slog's compilation to high-performance, data-parallel kernels differentiates it from Egglog, which instead employs lazy rebuilding. Our subjective experience comparing Slog with Egglog is that Slog is faster in applications where materialization-overhead is not a bottleneck. We plan to study the synthesis of Slog and e-graphs as future work.

Incremental Datalog. A significant amount of recent interest focuses on incremental Datalog, extending Datalog's semi-naïve evaluation strategy to interesting domains [2, 16, 66, 73], especially focusing on generalizing Datalog's semantics in ways in which are incremental by design. Relevant systems include timely dataflow [73], RDFox [74], and DBSP [16]. Compared to these systems, $\mathcal{DL}^{\exists!}$ does not support negation or an inverse operator for elements of $\mathcal{V}$ and $\mathcal{S}$, instead preferring to be obviously-monotonic by construction. While $\mathcal{DL}^{\exists!}$ and Slog do use semi-naïve evaluation, they do not expose the engine's data structures in a stream-based manner. We leave to further study the connection between Slog with differential dataflow; we expect it is possible to compile $\mathcal{DL}^{\exists!}$ to DBSP, but note that our current backend is built on MPI and has the potential to leverage technologies such as InfiniBand.

Distributed Datalog. There have been significant efforts to scale Datalog-like languages to large clusters of machines. For example, RDFox [74], BigDatalog [91], Distributed SociaLite [87], Myria [49], and Radlog [48] all run on Apache Spark clusters. Slog differs from these systems in two primary ways. First, compared to Slog's MPI-based implementation, Apache Spark's framework-imposed overhead is increasingly understood to be a bottleneck in scalable data analytics applications, with several authors noting order-of-magnitude improvements when switching from Spark to MPI [4, 59, 83]. Second, none of the aforementioned systems support first-class facts. We elide detailed comparison against these systems due to space; we found Slog was faster and more scalable than Radlog, BigDatalog, and similar systems in our limited explorations.

7 CONCLUSION

We presented $\mathcal{DL}^{\exists!}$, a Datalog with first-class facts. In $\mathcal{DL}^{\exists!}$, facts are identified by a symbolic representation in the form of a Skolem term. This methodology extends Datalog with the ability to introspect upon, compute with, and construct new fact identities. We rigorously define a semantics of $\mathcal{DL}^{\exists!}$, presenting its semantics as a variant of the restricted chase which materializes Skolem terms, essentially giving facts an algebraic identity. The fact that rule heads cannot be unified with the body forbids the construction of cyclic facts, and thus yields a semi-decidable semantics amenable to bottom-up implementation via a simple extension of modern-generation Datalog engines. Our higher-level language $\mathcal{DL}_{\mathcal{S}}$ offers an ergonomic syntax, enabling a more direct transliteration of functional programs and natural-deduction-style rules into $\mathcal{DL}^{\exists!}$. We defined $\mathcal{DL}_{\mathcal{S}}$ via a syntax-directed translation into $\mathcal{DL}^{\exists!}$, and showed its model-theoretic and fixpoint-based semantics, formally the equivalence of these semantics in Isabelle/HOL.

In our implementation, we generalize $\mathcal{DL}^{\exists!}$ to full-fledged engine, Slog, in which we have implemented a wide breadth of applications, including various forms of provenance (§5.1) and structural abstract interpretation (§3.5). Slog exploits the uniqueness properties of $\mathcal{DL}^{\exists!}$ to ensure a massively-parallel implementation which materializes first-class facts using a communication-avoiding strategy which yields highly-scalable implementations in practice, compounding the algorithmic benefits of $\mathcal{DL}^{\exists!}$. Our experiments speak to the promise of our approach: Slog beats all other comparison engines in our benchmarks when run at sufficient scale, and our strong-scaling runs (context-sensitive points-to analysis of Linux, §5.3) show satisfactory scalability up to 1k cores. Slog is available at:

https://github.com/harp-lab/slog-lang1

ACKNOWLEDGEMENTS

This work was funded in part by NSF PPoSS large grants CCF-2316159 and CCF-2316157. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[2] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. Fixing Incremental Computation. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 525–552.

[3] Nada Amin and Tiark Rompf. 2017. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

[4] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. 2017. Bridging the Gap between HPC and Big Data Frameworks. *Proc. VLDB Endow.* 10, 8 (apr 2017), 901–912. https://doi.org/10.14778/3090163.3090168

[5] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting DOOP to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 25–30.

[6] Molham Aref, Balder Ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

[7] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A toolkit for query answering with existential rules. In *Rule Technologies: Foundations, Tools, and Applications: 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings 9*. Springer, 328–344.

[8] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artificial Intelligence* 175, 9-10 (2011), 1620–1654.

[9] Teodoro Baldazzi, Luigi Bellomarini, and Emanuel Sallinger. 2023. Reasoning over Financial Scenarios with the Vadalog System. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, Vol. 26. OpenProceedings. org, 782–791.

[10] Luigi Bellomarini, Davide Benedetto, Matteo Brandetti, and Emanuel Sallinger. 2022. Exploiting the Power of Equality-Generating Dependencies in Ontological Reasoning. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 3976–3988. https://doi.org/10.14778/3565838.3565850

[11] Luigi Bellomarini, Davide Benedetto, Matteo Brandetti, and Emanuel Sallinger. 2022. Exploiting the power of equality-generating dependencies in ontological reasoning. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3976–3988.

[12] Pierre Bourhis, Marco Manna, Michael Morak, and Andreas Pieris. 2016. Guarded-based disjunctive tuple-generating dependencies. *ACM Transactions on Database Systems (TODS)* 41, 4 (2016), 1–45.

[13] Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. 2023. Modular Abstract Definitional Interpreters for WebAssembly. In *European Conference on Object-Oriented Programming (ECOOP 2023)*.

[14] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[15] J. Bruck, , S. Kipnis, E. Upfal, and D. Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 11 (Nov 1997), 1143–1156. https://doi.org/10.1109/71.642949

[16] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (mar 2023), 1601–1614. https://doi.org/10.14778/3587136.3587137

[17] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 316–330.

[18] Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. 2024. The Complexity of Why-Provenance for Datalog Queries. *Proc. ACM Manag. Data* 2, 2, Article 83 (may 2024), 16 pages. https://doi.org/10.1145/3651146

[19] Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. 2024. Computing the Why-Provenance for Datalog Queries via SAT Solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 10459–10466.

[20] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 77–86.

[21] David Carral, Irina Dragoste, and Markus Krötzsch. 2017. Restricted Chase (Non) Termination for Existential Rules with Disjunctions.. In *IJCAI*. 922–928.

[22] David Carral, Irina Dragoste, Markus Krötzsch, and Christian Lewe. 2019. Chasing sets: how to use existential rules for expressive reasoning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 1624–1631.

[23] David Carral, Irina Dragoste, and Markus Krötzsch. 2017. Restricted Chase (Non)Termination for Existential Rules with Disjunctions. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 922–928. https://doi.org/10.24963/ijcai.2017/128

[24] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.

[25] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.

[26] Cognitect, Inc. [n.d.]. Datomic: A Distributed Deductive Database in Clojure. https://www.datomic.com/. accessed: 9-22-2024.

[27] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

[28] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[29] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 269–282.

[30] Yingwei Cui and Jennifer Widom. 2000. Lineage tracing in a data warehousing system. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 683–684.

[31] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (jun 2000), 179–227. https://doi.org/10.1145/357775.357777

[32] David Darais, Nicholas Labich, Phúc C Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–25.

[33] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[34] Ali Elhalawati, Markus Krötzsch, and Stephan Mennicke. 2022. An existential rule framework for computing why-provenance on-demand for datalog. In *International Joint Conference on Rules and Reasoning*. Springer, 146–163.

[35] Agneta Eriksson and Anna-Lena Johansson. 1985. *Neat explanation of proof trees.* Uppsala University, Computing Science Department, Uppsala Programming ....

[36] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. 2015. Fpsolve: A generic solver for fixpoint equations over semirings. *International Journal of Foundations of Computer Science* 26, 07 (2015), 805–825.

[37] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the Bruck Algorithm for Non-Uniform All-to-All Communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3502181.3531468

[38] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh Patel. 2018. Scaling-up in-memory datalog processing: Observations and techniques. *arXiv preprint arXiv:1812.03975* (2018).

[39] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. 2005. Explanations and proof trees. In *International Symposium on Explanation-Aware Computing, ExaCt 2005*. AAAI Press, 76–85.

[40] Message P Forum. 1994. MPI: A message-passing interface standard.

[41] Kimball Germane and Michael D Adams. 2020. Liberate Abstract Garbage Collection from the Stack by Decomposing the Heap. In *European Symposium on Programming*. Springer, Cham, 197–223.

[42] Kimball Germane and Jay McCarthy. 2021. Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–28.

[43] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP '16)*. ACM, New York, NY, USA, 407–420. https://doi.org/10.1145/2951913.2951936

[44] Thomas Gilray and Sidharth Kumar. 2019. Distributed Relational Algebra at Scale. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 12–22. https://doi.org/10.1109/HiPC.2019.00014

[45] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. 2021. Compiling Data-Parallel Datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 23–35. https://doi.org/10.1145/3446804.3446855

[46] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

[47] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (, Beijing, China,) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

[48] Jiaqi Gu, Yugo H Watanabe, William A Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater power and performance for big data analytics with recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 467–484. https://doi.org/10.1145/3299869.3324959

[49] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 881–884. https://doi.org/10.1145/2588555.2594530

[50] Matthew Hennessy. 1990. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc.

[51] Alex Ivliev, Stefan Ellmauthaler, Lukas Gerlach, Maximilian Marx, Matthias Meißner, Simon Meusel, and Markus Krötzsch. 2023. Nemo: First glimpse of a new rule engine. *arXiv preprint arXiv:2308.15897* (2023).

[52] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, Springer International Publishing, Cham, 422–430.

[53] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. Brie: A specialized trie for concurrent datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores* (Washington, DC, USA) *(PMAM'19)*. ACM, New York, NY, USA, 31–40.

[54] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th symposium on principles and practice of parallel programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 327–339.

[55] Gilles Kahn. 1987. Natural semantics. In *Annual symposium on theoretical aspects of computer science*. Springer, 22–39.

[56] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. 2014. The Incredible ELK: From Polynomial Procedures to Efficient Reasoning with EL Ontologies. *Journal of automated reasoning* 53, 1 (2014), 1–61.

[57] Andrew W. Keep and R. Kent Dybvig. 2013. A Nanopass Framework for Commercial Compiler Development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 343–350. https://doi.org/10.1145/2500365.2500618

[58] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation* 20, 3 (2007), 199–207.

[59] Deepa S Kumar and M Abdul Rahman. 2017. Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis. *Journal of Computer Science* 13, 12 (Dec 2017), 781–794. https://doi.org/10.3844/jcssp.2017.781.794

[60] R. Kumar, A. Mamidala, and D. K. Panda. 2008. Scaling alltoall collective on multi-core systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.

[61] Sidharth Kumar, Cameron Christensen, JohnA. Schmidt, Peer-Timo Bremer, Eric Brugger, Venkatram Vishwanath, Philip Carns, Hemanth Kolla, Ray Grout, Jacqueline Chen, Martin Berzins, Giorgio Scorzelli, and Valerio Pascucci. 2014. Fast Multiresolution Reads of Massive Simulation Datasets. In *Supercomputing*, JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer (Eds.). Lecture Notes in Computer Science, Vol. 8488. Springer International Publishing, 314–330. https://doi.org/10.1007/978-3-319-07518-1_20

[62] Sidharth Kumar and Thomas Gilray. 2020. Load-balancing parallel relational algebra. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*. Springer, 288–308.

[63] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. 2012. Efficiently computable Datalog$^{\exists}$ programs. In *Thirteenth international conference on the principles of knowledge representation and reasoning*.

[64] David Maier, Alberto O Mendelzon, and Yehoshua Sagiv. 1979. Testing implications of data dependencies. *ACM Transactions on Database Systems (TODS)* 4, 4 (1979), 455–469.

[65] Per Martin-Löf. 1996. On the Meanings of the Logical Constants and the Justifications of the Logical Laws. *Nordic Journal of Philosophical Logic* 1, 1 (1996), 11–60.

[66] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow.. In *CIDR*.

[67] Matthew Might. 2010. Abstract interpreters for free. In *International Static Analysis Symposium (SAS '10)*. Springer, 407–421.

[68] Matthew Might and Panagiotis Manolios. 2009. A posteriori soundness for non-deterministic abstract interpretations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 260–274.

[69] Matthew Might and Olin Shivers. 2008. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming* 18, 5-6 (2008), 821–864.

[70] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 305–315.

[71] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of datalog materialisations revisited. *Artificial Intelligence* 269 (2019), 76–136.

[72] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 28.

[73] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[74] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *The Semantic Web - ISWC 2015*, Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.). Springer International Publishing, Cham, 3–20.

[75] Phuc C Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft contract verification for higher-order stateful programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 51.

[76] André Pacak and Sebastian Erdweg. 2019. Generating incremental type services. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (Athens, Greece) *(SLE 2019)*. Association for Computing Machinery, New York, NY, USA, 197–201. https://doi.org/10.1145/3357766.3359534

[77] André Pacak and Sebastian Erdweg. 2022. Functional programming with Datalog. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

[79] Scott Parker, Vitali Morozov, Sudheer Chunduri, Kevin Harms, Chris Knight, and Kalyan Kumaran. 2017. *Early Evaluation of the Cray XC40 Xeon Phi System 'Theta'at Argonne*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[80] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[81] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

[82] Gordon D Plotkin. 1981. Structural operational semantics. *Aarhus University, Denmark* (1981), 20–23.

[83] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. 2015. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science* 53 (2015), 121–130. https://doi.org/10.1016/j.procs.2015.07.286 INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[84] John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. 717–740.

[85] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 264 (oct 2023), 26 pages. https://doi.org/10.1145/3622840

[86] David S Scott. 1991. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*. IEEE Computer Society, 398–399.

[87] Jiwon Seo, Stephen Guo, and Monica S Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 278–289.

[88] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

[89] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 399–410.

[90] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, PA.

[91] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[92] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2023. GDlog: A GPU-Accelerated Deductive Engine. arXiv:2311.02206 [cs.DB]

[93] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[94] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics* 5, 2 (1955), 285–309.

[95] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (Feb. 2005), 49–66.

[96] Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order symbolic execution via contracts. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 537–554.

[97] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*. 135–144.

[98] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. 2016. Column-oriented datalog materialization for large knowledge graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[99] Patrick Valduriez and Setrag Khoshafian. 1988. Parallel Evaluation of the Transitive Closure of a Database Relation. *Int. J. Parallel Program.* 17, 1 (Feb. 1988), 19–42.

[100] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

[101] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Harry Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), 389–404. https://doi.org/10.1145/3037697.3037744

[102] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–32.

[103] Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–28.

[104] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[105] Rohit Zambre, Damodar Sahasrabudhe, Hui Zhou, Martin Berzins, Aparna Chandramowlishwaran, and Pavan Balaji. 2021. Logically Parallel Communication for Fast MPI+ Threads Applications. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[106] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (jan 2022), 22 pages. https://doi.org/10.1145/3498696

[107] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and J. Guo. 2020. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 673–684.