Certifying Standard and Stratified Datalog Inference Engines in SSReflect

Véronique Benzaken, Évelyne Contejean, Stefania Dumbrava

Certifying Standard and Stratified Datalog Inference Engines in SSReflect

Véronique Benzaken[1], Évelyne Contejean[2], and Stefania Dumbrava[3]

[1] Université Paris Sud, LRI, France
[2] CNRS, LRI, Université Paris Sud, France
[3] LIRIS, Université Lyon 1, France

Abstract. We propose a SSReflect library for logic programming in the Datalog setting. As part of this work, we give a first mechanization of standard Datalog and of its extension with stratified negation. The library contains a formalization of the model theoretical and fixpoint semantics of the languages, implemented through bottom-up and, respectively, through stratified evaluation procedures. We provide corresponding soundness, termination, completeness and model minimality proofs. To this end, we rely on the Coq proof assistant and SSReflect. In this context, we also construct a preliminary framework for dealing with stratified programs. We consider this to be a necessary first step towards the certification of security-aware data-centric applications.

1 Introduction

Datalog [7] is a deductive language capturing the function-free fragment of Horn predicate logic. Syntactically a subset of Prolog [22], Datalog has the advantage of guaranteed termination (in PTIME [33]). Expressivity-wise, it extends relational algebra/calculus with recursion; also, it allows for computing transitive closures and, generally, for compactly formulating graph traversals and analytics. Comparatively, more popular query languages, such as SQL, XPath, or SPARQL, are either not capable of expressing these directly/inherently or do so with limitations. Given the present ubiquity of interconnected data, seamlessly supporting such recursive queries has regained relevance. For example, these are key to Web infrastructure, being used by webcrawlers and PageRank algorithms. Efficiently querying and reasoning about graph topologies is, in fact, fundamental in a variety of areas, including, but not limited to: the Semantic Web; social, communication and biological networks; and geographical databases.

Due to its purely declarative nature and simplicity (few primitives), Datalog lends itself particularly well to domain-specific extensions. As surveyed in the literature [28,2], multiple additions to its core language have been introduced, e.g, with negation, existentials, aggregates, functions, updates, etc. Indeed, Datalog can be seen as the lingua franca for a plethora of custom query languages, e.g, Starlog [25], Overlog [24], Netlog [17], DATALOG± [6], SociaLite [30], LogiQL [4], etc. In a recent resurge of interest [1], marked by the *Datalog 2.0 Workshop*, such tailored versions of Datalog found new applications in data integration, security [10], program analysis [34], cloud computing, parallel and distributed programming [18], etc. An overview is given in [19].

These applications have not only sparked interest in the academic setting, but also in the industry one. Indeed, commercial Datalog engines have started to gain popularity, with LogicBlox [23], Google's Yedalog [8], Datomic [9], Exeura [12], Seemle [29], and Lixto [16], as prominent examples. Moreover, their scope has extended to include safety-critical, large-scale use cases. A case in point is the LogicBlox platform, which underpins high-value web retail and insurance applications. Its Datalog-based engine unifies the modern enterprise software stack (encompassing bookkeeping, analytics, planning, and forecasting) and runs with impressive efficiency [5]. Also, more recently, Datalog has been proposed as tool for automating the verification of railway infrastructure high-security regulations against its CAD design [26].

We argue that, given the role Datalog is starting to play in data-centric and security-sensitive applications, obtaining the strong guarantees Coq certification provide is an important endeavour. We envisage a methodology aimed at ultimately certifying a realistic Datalog engine by *refinement*. This would encompass: 1) a high-level formalization suitable for proof-development and, thus, employing more inefficient algorithms, 2) a mechanization of the real-world engine implementation, and 3) (refinement) proofs of their extensional equivalence.

This paper describes the first necessary step towards realizing this vision. As such, we propose a deep specification of a stratified Datalog inference engine in the SSReflect extension [15] of the Coq proof-assistant [27]. With respect to the scope of our formalization, the chosen fragment is the one used by LogicBlox and it is the most expressive one that retains termination[4]. Our chosen evaluation heuristic is bottom-up, as ensuring that top-down/more optimized heuristics do not get stuck in infinite loops is harder to establish. Also, this allows us to modularly extend and reuse our standard Datalog inference engine in the stratified setting. We do envisage supporting, for example, magic-sets rewriting.

The choice of using SSReflect is due to the fact that the model-theoretic semantics of Datalog is deeply rooted in *finite model theory*. To quote [21]: "*For many years, finite model theory was viewed as the backbone of database theory, and database theory in turn supplied finite model theory with key motivations and problems. By now, finite model theory has built a large arsenal of tools that can easily be used by database theoreticians without going to the basics*". The Mathematical Components library[5], built on top of SSReflect, is especially well-suited for our purposes, as it was the basis of extensive formalizations of finite model theory, in the context of proving the Feit-Thompson theorem [14], central to finite group classification. Moreover, as detailed next, our proof-engineering efforts were much greatly by our reuse of the `fintype`, `finset` and `bigop` libraries.

Contributions Our key modeling choice is properly setting up the base types to make the most of the finite machinary of SSReflect. Heavily relying on type finiteness ensures desirable properties, such as decidability. As every Datalog program has a finite model [2], i.e, its Herbrand Base (Section 2.1), this does not restrict generality. The paper's contributions are:

[4] Arithmetic predicates and skolem function destroy this guarantee.

[5] `http://math-comp.github.io/math-comp/`

1. a certified "positive" inference engine for standard Datalog: We give a scalable formalization of the syntax, semantics and bottom-up inference of Datalog. The latter consists of mechanizing a matching algorithm for terms, atoms and clause bodies and proving corresponding soundness and completeness results. We formally characterize the engine, by establishing *soundness*, *termination*, *completeness* and *model minimality*, based on *monotonicity*, *boundedness* and *stability* proofs.
2. a certified "negative" inference engine for stratified Datalog: We extend the syntax and semantics of Datalog with negation and mechanize its stratified evaluation. We model program stratification and "slicing", embed negated literals as flagged positive atoms and extend the notion of an interpretation to that of a "complemented interpretation". The crux of stratified evaluation is the reuse of the "positive" engine, for each program "slice". When formally characterizing the "negative engine", this required us to precisely identify additionaly properties, i.e, *incrementality* and *modularity*, and to correspondingly extend the previous library. We establish *soundness*, *termination*, *completeness* and *model minimality*.

Lastly, we extract our standard Datalog engine in OCaml as a proof-of-concept.

Organization The paper is organized as follows. In Section 2, we give a concise theoretical summary of standard and stratified Datalog. In Sections 3 and 4, we present the corresponding SSReflect inference engine mechanizations. Section 5 describes related work. We conclude in Section 6.

2 Preliminaries

We review the theory of standard and stratified Datalog in Sections 2.1 and 2.2.

2.1 Standard Datalog

Syntax Given the sets $\mathcal{V}$, $\mathcal{C}$ and $\mathcal{P}$ of variables, constants and predicate symbols, a program is a *finite* collection of clauses, as captured by the grammar:

Programs	P	$::=$	$C_1, \ldots, C_k$
Clauses	C	$::=$	$A_0 \leftarrow A_1, \ldots, A_m$
Atoms	A	$::=$	$p(\vec{t})$, where $p \in \mathcal{P}$ is denoted $sym(A)$ and has arity $ar(p) = \lvert\vec{t}\rvert$ [6]
Terms	t	$::=$	$x \in \mathcal{V} \mid c \in \mathcal{C}$

A clause is a sentence separating hypotheses *body* atoms from the conclusion *head* atom. Clauses allow inferring new facts (true assertions) from existing ones. The below restriction ensures *finitely* many facts are inferred.

Definition 1 (Safety). *A standard Datalog program is* safe *iff all of its clauses are* safe*, i.e, all of their head variables appear in their body.*
Consequently, safe program facts are ground [7]*.*

[6] Term sequences $t_1, \ldots, t_n$ are abbreviated as $\vec{t}$ and $\lvert\vec{t}\rvert = n$ denotes their length.
[7] We call language constructs that are variable-free, *ground* and, otherwise, *open*.

Semantics Let $\mathbf{B}(P)$ be the Herbrand Base of P, i.e, the ground atom set built from its predicates and constants. By the Herbrand semantics, an *interpretation* I is a subset of $\mathbf{B}(P)$. For a valuation ν, mapping bound clause variables to program constants[8], and a clause C equal to $p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}), \ldots, p_m(\vec{t_m})$, the *clause grounding*[9] νC is $p_0(\nu\vec{t_0}) \leftarrow p_1(\nu\vec{t_1}), \ldots, p_m(\nu\vec{t_m})$. Note that variables are implicitly universally quantified and, hence, occurences of the same variable in C have to be instantiated in the same way by ν. C is then satisfied by I iff, *for all* valuations ν, if $\{p_1(\nu\vec{t_1}), \ldots, p_m(\nu\vec{t_m})\} \subseteq I$ then $p_0(\nu\vec{t_0}) \in I$. I is a *model* of P iff all clauses in P are satisfied by I. The *intended semantics* of P is $\mathbf{M_P}$, its *minimal model* w.r.t set inclusion. This model-theoretic semantics indicates *when* an interpretation is a model, but not *how* to construct such a model. Its *computational* counterpart centers on the *least fixpoint* of the following operator.

Definition 2 (The T_P Consequence Operator). *Let P be a program and I an interpretation. The T_P operator is the set of program* consequences *F:*
$T_P(I) = \{F \in \mathbf{B}(P) \mid F \in I \vee F = head(\nu C),\ for\ C \in P \wedge body(\nu C) \subseteq I\}$.

Definition 3 (Fixpoint Evaluation). *The iterations of the T_P operator are: $T_P \uparrow 0 = \emptyset$, $T_P \uparrow (n+1) = T_P(T_P \uparrow n)$. Since T_P is monotonous and bound by $\mathbf{B}(P)$, the Knaster-Tarski theorem [31] ensures $\exists\omega, T_P \uparrow \omega = \bigcup_{n \geq 0} T_P \uparrow n$, where $T_P \uparrow \omega = lfp(T_P)$. The* fixpoint evaluation *of P is thus defined as $lfp(T_P)$.*
Note that, by van Emden and Kowalski [32], $lfp(T_P) = \mathbf{M_P}$.

Example 1. Let $P = \begin{cases} e(1,3).\ e(2,1).\ e(4,2).\ e(2,4). \\ t(X,Y) \leftarrow e(X,Y). \\ t(X,Y) \leftarrow e(X,Z), t(Z,Y). \end{cases}$
$T_P \uparrow 0 = \emptyset$; $T_P \uparrow 1 = \{e(1,3), e(2,1), e(4,2), e(2,4)\}$; $T_P \uparrow 2 = T_P \uparrow 1 \cup \{t(1,3), t(2,1), t(4,2), t(2,4)\}$; $T_P \uparrow 3 = T_P \uparrow 2 \cup \{t(2,3), t(4,1), t(4,4), t(2,2)\}$. The minimal model of P is $\mathbf{M_P} = lfp(T_P) = T_P \uparrow 4 = T_P \uparrow 3 \cup \{t(4,3)\}$.

2.2 Stratified Datalog

Syntax Adding stratified negation amounts to extending the syntax of standard Datalog, by introducing *literals* and adjusting the definition for clauses.

$$\begin{array}{rrcl} \textit{Clauses} & C & ::= & A \leftarrow L_1, \ldots, L_m \\ \textit{Literals} & L & ::= & A \mid \neg A \end{array}$$

Definition 4 (Predicate Definitions). *Let P be a program. The* definition *$def(p)$ of program predicate $p \in \mathcal{P}$ is $\{C \in P \mid sym(head(C)) = p\}$.*

Definition 5 (Program Stratification and Slicing).
Let P be a program with clauses C of the form $H \leftarrow L_1, \ldots, L_k, \neg L_{k+1}, \ldots, \neg L_l$, where $body^+(C) = \{L_1, \ldots, L_k\}$ and $body^-(C) = \{L_{k+1}, \ldots, \neg L_l\}$. Consider a mapping $\sigma : \mathcal{P} \to [1,n]$, such that: 1) $\sigma(sym(L_j)) \leq \sigma(sym(H))$, for $j \in [1,k]$, and 2) $\sigma(sym(L_j)) < \sigma(sym(H))$, for $j \in [k+1,l]$. σ induces a partitioning [10]

[8] The set of program constants is also called its *active domain*, denoted $adom(P)$.
[9] Also called *clause instantiation*.
[10] $\sqcup$ denotes the pairwise disjoint set union.

$P = \bigsqcup_{j \in [1,n]} P_{\sigma_j}$ *with* $\sigma_j = \{p \in \mathcal{P} \mid \sigma(p) = j\}$ *and* $P_{\sigma_j} = \bigcup_{p \in \sigma_j} def(p)$. *We have that, for* $C \in P_{\sigma_j}$*: 1) if* $p \in \{sym(L) \mid L \in body^-(C)\}$*, then* $def(p) \subseteq \bigcup_{1 \leq k < j} P_{\sigma_k}$*, and 2) if* $p \in \{sym(L) \mid L \in body^+(C)\}$*, then* $def(p) \subseteq \bigcup_{1 \leq k \leq j} P_{\sigma_k}$.

We call P stratified*;* σ*, a* stratification*;* σ_j*, a* stratum*; the set* $\{P_{\sigma_1}, \ldots, P_{\sigma_n}\}$*, a* program slicing [11] *and* P_{σ_j}*, a* program slice*, henceforth denoted* P_j.

Stratification ensures program slices P_j are *semipositive* programs [2] that can be evaluated *independently*. Indeed, checking if their negated atoms belong to some interpretation I is equivalent to checking that their positive counterparts belong to the *complement* of I w.r.t the Herbrand Base $\mathbf{B}(P_j)$.

Semantics The model of a stratified Datalog program is given by the stepwise, bottom-up computation of the least fixpoint model for each of its slices.

Definition 6 (Stratified Evaluation). *For* $P = P_1 \sqcup \ldots \sqcup P_n$*, the model* [12]*,* $M_n = T_{P_n} \uparrow \omega(M_{n-1})$*, where* $M_j = T_{P_j} \uparrow \omega(M_{j-1})$*,* $j \in [2, n]$*,* $M_1 = T_{P_1} \uparrow \omega(\emptyset)$.

Example 2. Let $P = \begin{cases} q(a).\ s(b).\ t(a).\ r(X) \leftarrow t(X). \\ p(X) \leftarrow \neg q(X), r(X). \\ p(X) \leftarrow \neg t(X), q(X). \\ q(X) \leftarrow s(X), \neg t(X). \end{cases}$ for which a *stratification* $\sigma(s) = 1$, $\sigma(t) = 1$, $\sigma(r) = 1$, $\sigma(q) = 2$, $\sigma(p) = 3$, with the *strata* $\sigma_1 = \{s, t, r\}$, $\sigma_2 = \{q\}$, $\sigma_3 = \{p\}$, induces the partitioning $P = P_1 \sqcup P_2 \sqcup P_3$, with the *slices* $P_1 = \begin{cases} s(b).\ t(a). \\ r(X) \leftarrow t(X). \end{cases}$ $P_2 = \begin{cases} q(a). \\ q(X) \leftarrow s(X), \neg t(X). \end{cases}$ $P_3 = \begin{cases} p(X) \leftarrow \neg q(X), r(X). \\ p(X) \leftarrow \neg t(X), q(X). \end{cases}$
$M_1 = T_{P_1} \uparrow \omega(\emptyset) = \{r(a), s(b), t(a)\}$; $M_2 = T_{P_2} \uparrow \omega(M_1) = M_1 \cup \{q(a), q(b)\}$;
$\mathbf{M_P} = M_3 = T_{P_3} \uparrow \omega(M_2) = M_2 \cup \{p(b)\} = \{r(a), s(b), t(a), q(a), q(b), p(b)\}$.

3 A Mechanized Standard Datalog Engine

In Section 3.1, we present our formalization of the syntax and semantics of standard Datalog. Next, in Section 3.2, we detail the bottom-up evaluation heuristic of its inference engine. We formally characterize the engine in Section 3.3.

3.1 Formalizing Standard Datalog

Syntax We assume the `stype` and `ctype` finite types for predicate *symbols* and *constants*, as well as an `arity` finitely-supported function.

```
Variables (stype ctype : finType) (arity : {ffun stype → nat}).
```

Terms are encoded by an inductive joining 1) variables, of ordinal type `'I_n`, bound by a computable maximal value `n`, and 2) constants.

[11] A program can have multiple stratifications.
[12] As proven by Apt[3], M_n is *independent from the choice of stratification.*

```
Inductive term : Type := Var of 'I_n | Val of constant.
```

To avoid redundant case analyses, we henceforth distinguish between *ground* and *open* (non-ground) atoms and clauses. Intuitively, this dichotomy is desirable as the former are primitives of the *semantics*, while the latter, of the *syntax*. As such, *ground atoms* are modeled with `gatom` records, joining the `rgatom` base type and the boolean well-formedness condition `wf_rgatom`. The first packs a symbol and a list of *constants*, while the second ensures symbol arity and argument size match. Note that, as we set up the `gatom` subtyping predicate to be inherently proof-irrelevant, checking ground atom equality can be conveniently reduced to checking the equality of their underlying base types. *Atoms* are encoded similarly, except that their base type packs a *term* list instead.

```
Inductive  rgatom := RawGAtom of symtype & seq constant.
Definition wf_rgatom rga := size (arg rga) == arity (sym rga).
Structure  gatom := GAtom {rga :> rgatom; _ : wf_rgatom rga}.
```

(Ground) clauses pack a distinguished (ground) atom and a (ground) atom list. *Programs* are clause lists. The safety condition formalization mirrors Definition 1.

Semantics An *interpretation* `i` is a finite set of ground atoms. Note that, since its type, `interp`, is finite, the latter has a lattice structure, whose top element, `setT`, is the set of all possible ground atoms. The *satisfiability* of a ground clause `gcl` w.r.t `i` is encoded by `gcl_true`. As in Section 2.1, we define `i` to be a *model* of a program `p`, if, for all grounding substitutions ν, it satisfies all corresponding clause instantiations. We discuss the encoding of grounding substitutions next.

```
Notation interp := {set gatom}.
Definition gcl_true gcl i := (* i satisfies gcl *)
  all (mem i) (body_gcl gcl) ==> (head_gcl gcl ∈ i).
Definition prog_true  p i := (* i is a model of p *)
  ∀ν : gr, all (fun cl ⇒ gcl_true (gr_cl ν cl) i) p.
```

3.2 Mechanizing the Bottom-up Evaluation Engine

The inference engine iterates the logical consequence operator from Definition 2. To build a model of an input program, it maintains a current "candidate model" interpretation, which it iteratively tries to "repair". The repair process first identifies clauses that violate satisfiability, i.e, whose ground instance bodies are in the current interpretation, but whose heads are not. The current interpretation is then "fixed", adding to it the missing facts, i.e, the head groundings. This is done by a *matching algorithm*, incrementally constructing *substitutions* that homogeneously instantiate all clause body atoms to "candidate model" facts. As safety ensures all head variables appear in the body, these substitutions are indeed grounding. Hence, applying them to the head produces *new facts*. Once the current interpretation is "updated" with all facts inferrable in one forward chain step, the procedure is repeated, until a fixpoint is reached. We prove this to be a *minimal model* of the input program. As outlined, the mechanization of the engine centers around the encoding of *substitutions* and of *matching* functions.

Groundings and Substitutions Following a similiar reasoning to that in Section 3.1, we define a separate type for grounding substitutions (groundings). Both groundings and substitutions are modeled as finitely-supported functions from variables to constants, except for the latter being partial [13].

```
Definition gr  := {ffun 'I_n → constant}.
Definition sub := {ffun 'I_n → option constant}.
```

We account for the engine's gradual extension of substitutions, by introducing a *partial ordering*[14] over these. To this end, using finitely-supported functions was particularly conveninent, as they can be used both as functions and as lists of bindings. We say a substitution σ_2 *extends* a substitution σ_1, if all variables bound by σ_1 appear in σ_2, bound to the same values. We model this predicate as `sub_st`[15] and the *extension* of a substition σ, as the `add` finitely-supported

```
Definition sub_st σ₁ σ₂ := (* henceforth denoted as σ₁ ⊆ σ₂ *)
  [∀v : 'I_n, if σ₁ v is Some c then (v, c) ∈ σ₂ else true].
Definition add σ v c :=
  [ffun u ⇒ if u == v then Some c else σ u].
```

Term Matching Matching a term t to a constant d under a substitution σ, will either: 1) return the *input substitution*, if t or $\sigma\ t$ equal d, 2) return the *extension* of σ with the corresponding binding, if t is a variable not previously bound in σ, or 3) *fail*, if t or $\sigma\ t$ differ from d.

```
Definition match_term d t σ : option sub :=
  match t with
   | Val e ⇒ if d == e then Some σ else None
   | Var v ⇒ if σ v is Some e
             then (if d == e then Some σ else None)
             else Some (add σ v d)
  end.
```

Atom Matching We define the `match_atom` and `match_atom_all` functions that return substitutions and, respectively, substitution sets, instantiating an atom to a ground atom and, respectively, to an interpretation. To compute the substitution matching a raw-atom `ra` to a ground one `rga`, we first check their symbols and argument sizes agree. If such, we extend the initial substitution σ, by iterating term matching over the itemwise pairing of their terms `zip arg2 arg1`. As term matching can fail, we wrap the function with an option binder extracting the corresponding variable assignations, if any. Hence, `match_raw_atom` is a monadic option fold that either fails or returns substitutions extending σ. Atom matching equals raw atom matching, by coercion to `raw_atom`.

```
Definition match_raw_atom rga ra σ : option sub :=
  match ra, rga with RawAtom s1 arg1, RawGAtom s2 arg2 ⇒
    if (s1 == s2) && (size arg1 == size arg2)
    then foldl (fun acc p ⇒ obind (match_term p.1 p.2) acc)
               (Some σ) (zip arg2 arg1)
```

[13] Groundings can be coerced to substitutions and substitutions can be lifted to groundings, by padding with a default element `def`.
[14] We establish corresponding reflexivity, antisymmetry and transitivity properties.
[15] We cand use the boolean quantifier, as the ordinal type of variables is finite.

```
    else None
  end.
Definition match_atom σ a ga := match_raw_atom σ a ga.
```

Next, we compute the substitutions that can match an atom `a` to a fact in an interpretation `i`. This is formalized as the set of substitutions σ that belong to the set gathering all substitutions matching `a` to ground atoms `ga` in `i`.

```
Definition match_atom_all i a σ :=
  [set σ' | Some σ' ∈ [set match_atom ga a σ | ga ∈ i]].
```

While the `match_term` and `match_atom` functions are written as Gallina algorithms, we were able to cast the `match_atom_all` algorithm mathematically as: $\{\sigma' \mid \sigma' \in \{\texttt{match\_atom ga a } \sigma \mid \texttt{ga} \in \texttt{i}\}\}$. The function is key for expressing forward chain and fixpoint evaluation. Propagating its implementation, we could "reduce" soundness and completeness proofs to set theory ones. As such, it was particularly convenient we could rely on `finset` properties.

Body Matching The `match_body` function extends an initial substitution set `ssb` with bindings matching all atoms in the atom list `tl`, to an interpretation `i`. These are built using `match_atom_all` and *uniformly* extending substitutions matching each atom to `i`. We model this based on our definition of `foldS`, a monadic fold for the set monad. This iteratively composes the applications of a seeded function to all the elements of a list, flattening intermediate outputs.

```
Definition match_body i tl ssb := foldS (match_atom_all i) ssb tl.
```

The T_P Consequence Operator We model the logical consequences of a clause `cl` w.r.t an interpretation `i` as the set of *new* facts inferrable from `cl` by matching its body to `i`. Such a fact, `gr_atom_def def σ (head cl)`, is the head instantiation with the *grounding* matching substitution σ[16].

```
Definition emptysub : sub       := [ffun _ ⇒ None].
Definition cons_clause def cl i :=
  [set gr_atom_def def σ (head cl) |
       σ ∈ match_body i (body cl) [set emptysub]].
```

One-Step Forward Chain One inference engine iteration computes the set of *all* consequences inferrable from a program `p` and an interpretation `i`. This amounts to taking the union of `i` with all the program clause consequences. The encoding mirrors the mathematical expression $\texttt{i} \cup \bigcup_{\texttt{cl} \in \texttt{p}} \texttt{cons\_clause def i cl}$[17].

```
Definition fwd_chain def p i :=
  i ∪ \bigcup_(cl ← p) cons_clause def cl i.
```

3.3 Formal Characterization of the Bottom-Up Evaluation Engine

We first state the main intermediate theorems, leading up to the key Theorem 7. The first two results are established based on analogous ones for terms and atoms. We assume an intepretation `i` and a seed substitution set `ssb`.

[16] `gr_atom_def` lifts substitutions to groundings, by padding with the `def` constant.
[17] Thanks to using the `bigcup` operator from the SSReflect `bigop` library.

Theorem 1 (Matching Soundness). *Let* `tl` *be an atom list. If a substitution* σ *is in the output of* `match_body`*, extending* `ssb` *with bindings matching* `tl` *to* `i`*, then there exists a ground atom list* `gtl` *such that: 1)* `gtl` *is the instantiation of* `tl` *with* σ *and 2) all* `gtl` *atoms belong to* `i`*.* **Proof** by induction on `tl`.

Theorem 2 (Matching Completeness). *Let* `cl` *be a clause and* ν *a grounding* compatible *with any substitution* σ *in* `ssb`*. If* ν *makes the body of* `gcl` *true in* `i`*, then* `match_body` *outputs a compatible substitution smaller or equal to* ν*.*
Proof by induction on `tl`.

Theorem 3 (T_P Stability). *Let* `cl` *be a clause and* `i` *an interpretation satisfying it. The facts inferred by* `cons_clause` *are in* `i`*.* **Proof** by Theorem 1.

Theorem 4 (T_P Soundness). *Let* `cl` *be a* safe *clause and* `i` *an interpretation. If the facts inferred by* `cons_clause` *are in* `i`*, then* `i` *is a model of* `cl`*.*
Proof by Theorems 3 and 2.

Theorem 5 (Forward Chain Stability and Soundness). *Let* `p` *be a* safe *program. Then, an interpretation* `i` *is a model of* `p` *iff it is a* `fwd_chain` *fixpoint.*[18]
Proof by Theorems 3 and 4.

Theorem 6 (Forward Chain Fixpoint Properties). *The* `fwd_chain` *function is* monotonous*,* increasing *and* bound *by* $\mathbf{B}(P)$*.*
Proof by compositionality of set-theoretical properties.

Theorem 7 (Bottom-up Evaluation Soundness and Completeness). *Let* `p` *be a* safe *program. By iterating forward chain as many times as there are elements in* $\mathbf{B}(P)$*, the engine* terminates *and outputs a* minimal *model for* `p`*.*
Proof by Theorems 5 and 6, using a corrolary of the Knaster-Tarski result, as established in Coq by [11].

4 A Mechanized Stratified Datalog Engine

We overview the formalization of the syntax and semantics of stratified Datalog in Section 4.1. In Section 4.2 we present the mechanization of the stratified Datalog engine. We outline its formal characterization in Section 4.3.

4.1 Formalizing Stratified Datalog

Syntax We extend the syntax of positive Datalog with *literals*, reusing the definitions of ground/non-ground atoms. As before, we distinguish ground/non-ground literals and clauses. The former are encoded enriching ground/non-ground atoms with a boolean flag, marking whether they are negated.

```
Inductive glit := GLit of bool * gatom.
Inductive lit  := Lit  of bool * atom.
```

(Ground) clauses pack (ground) atoms and (ground) literal lists. The encodings of programs and their safety condition are the same as in Section 3.1.

[18] We state this as the `fwd_chainP` reflection lemma.

Semantics The only additions to Section 3.1 concern ground literals and clauses. The `glit_true` definition captures the fact that an interpretation `i` satisfies a ground literal `gl`, by casing on the latter's flag. If it is true, i.e the literal is positive, we check if the underlying ground atom is in `i`; otherwise, validity holds if the underlying ground atom is *not* in `i`. The definition for the satisfiability of a ground clause w.r.t `i` is analogous to that given in Section 3.1.

```
Definition glit_true i gl := if flag gl then gatom_glit gl ∈ i
                                        else gatom_glit gl ∉ i.
```

4.2 Mechanizing the Stratified Evaluation Engine

Stratification We model a stratification as a list of symbol sets, implicitly assuming the first element to be its lowest stratum. As captured by `is_strata_rec`, the characteristic properties mirror those in Definition 5. Namely, these are 1) *disjointness*: no two strata share symbols, 2) *negative-dependency*: stratum symbols can only refer to negated symbols in strictly lower strata, and 3) *positive-dependency*: stratum symbols only depend on symbols from lower or equal strata. We can give an effective, albeit inefficient algorithm for computing a stratification satisfying the above, by exploring the finite set of all possible program stratifications. Hence, we use the finite search infrastructure of SSReflect, i.e, the `[pick e : T | P e]` construct that, among all inhabitants of a finite type `T`, picks an element `e`, satisfying a predicate `P`.

Positive Embedding To enable the reuse of the forward chain operator in Section 3.2, we will embed the Coq representation of Datalog programs with stratified negation into that of standard Datalog programs, used by the positive engine. This is realized via functions that *encode*/*uncode* constructs *to*/*from* their "positive" counterparts; we denote these as $\ulcorner\cdot\urcorner/\llcorner\cdot\lrcorner$. To the end, we augment symbol types with a boolean flag, marking if the original atom is negated. For example, $\ulcorner s(a)\urcorner = (s, \top)(a)$, $\ulcorner\neg s(a)\urcorner = (s, \bot)(a)$, $\llcorner(\top, s)(a)\lrcorner = s(a)$ and $\llcorner(\bot, s)(a)\lrcorner = \neg s(a)$. We show literal encoding/uncoding are inverse w.r.t each other and, hence, injective, by proving the corresponding cancelation lemmas. For clauses, encoding is inverse to uncoding and, hence, injective, only when the flag of its encoded head atom is positive. This is expressed by a *partial* cancelation lemma; for the converse direction the cancelation lemma holds. Based on these injectivity properties, we prove Theorem 9.

Stratified Evaluation Let `p` be a program and `str`, a strata. The `evalp` stratified evaluation of `p` traverses `str`, accumulating the processed strata as $\text{str}_<$. It then computes the minimal model, cf. Theorem 7, for each induced program slice, $\text{p}_{\text{str}_<}$. The main modeling choice is to construct the *complemented intepretation* for $\text{p}_{\text{str}_<}$. This accounts for the all "negative" facts that hold, by absence from the current model. These will be collected, uncoded, in a second interpretation. The corresponding `cinterp` type is thus defined as an `interp` pairing. To bookkeep the accumulated strata $\text{str}_<$, we wrap `cinterp` and the symbol set type of $\text{str}_<$ in a *cumulative interpretation* type, `sinterp` .

```
Notation   cinterp := (interp * interp)%type.
Definition sinterp := (cinterp * {set symtype})%type.
```

At an intermediate step, having already processed $\text{str}_<$, we encode the `p_curr` program slice *up to* the current stratum `ss`. We feed it, together with the previous *complemented interpretation* `ci`, to the positive engine `pengine_step`. Since this operates on *positive interpretations* [19], we have to relate the two. As such, we define the `c2p_bij` bijection between them, i.e, mutually inverse functions `c2p` and `p2c`, and apply it to obtain the needed types. The positive engine iterates the forward chain operator, as many times as there are elements in the program bound `bp` [20]. It adds the facts inferable from the current stratum and outputs a *positive* intepretation. It does not add the *implicitly* true *negated* ground atoms.

```
Definition bp : pinterp := setT.
Definition pengine_step def (pp : pprogram) (ci : cinterp) : cinterp :=
  p2c (iter #|bp| (P.fwd_chain pdef pp) (c2p ci)).
```

Hence, the `ciC` complementation function augments `m_next.2` with the complement of `m_next.1` w.r.t `setT` [21]; the complement is filtered to ensure only atoms with symbols in `ss` are retained (see the encoding of `ic_ssym`).

```
Variables (def : constant) (p : program) (psf : prog_safe p).
Fixpoint evalp (str : strata) ((ci, str<) : sinterp) :=
match str with [::] ⇒ (ci, str<) | ss :: str> ⇒
  let p_curr := slice_prog p (str< ∪ ss) in
  let m_next := pengine_step def (encodep p_curr) ci in
  let m_cmpl := ciC ss m_next in evalp str> (m_cmpl, str< ∪ ss)
end.
```

The resulting `m_cmpl` is thus *well-complemented*. As encoded by `ci_wc`, the property states that, for *any* `ci` of `cinterp` type and *any* symbol set `ss`, the `ci` components partition the slicing of `setT` with `ss`, i.e, the set of all ground atoms with symbols in `ss`. The next strata, i.e, $\text{str}_>$, are processed by the recursive call.

Example 3. Revisiting Example 2, the slice encodings, marked by $\ulcorner\cdot\urcorner$, are:

$$\ulcorner\mathbf{P_1}\urcorner = \begin{cases}(\top,s)(b).\ (\top,t)(a).\\ (\top,r)(X) \leftarrow (\top,t)(X).\end{cases} \qquad \ulcorner\mathbf{P_2}\urcorner = \begin{cases}(\top,q)(a).\\ (\top,q)(X) \leftarrow (\top,s)(X),(\bot,t)(X).\end{cases}$$

$$\ulcorner\mathbf{P_3}\urcorner = \begin{cases}(\top,p)(X) \leftarrow (\bot,q)(X),(\top,r)(X).\\ (\top,p)(X) \leftarrow (\bot,t)(X),(\top,q)(X).\end{cases}$$

The positive engine computes the *minimal model* of $\ulcorner P_1\urcorner$: $M_1 = T_{\ulcorner P_1\urcorner} \uparrow \omega(\emptyset) = \{(\top,r)(a),(\top,s)(b),(\top,t)(a)\}$; complementing it w.r.t the Herband Base $\mathbf{B}(\ulcorner\mathbf{P_1}\urcorner)$ yields: $M_1 = \{(\bot,r)(b),(\bot,s)(a),(\bot,t)(b)\})$. Next, when passing the resulting *positive interpretation* $M_2 \cup M_2$ to the positive engine: $M_2 = T_{P_2} \uparrow \omega(M_1 \cup M_1) = M_1 \cup \{(\top,q)(a),(\top,q)(b)\}$. Its complement w.r.t $\mathbf{B}(\ulcorner\mathbf{P_2}\urcorner)$ is $M_2 = \{(\bot,r)(b),(\bot,s)(a),(\bot,t)(b)\}$. Finally, $M_3 = T_{P_3} \uparrow \omega(M_2 \cup M_2) = M_2 \cup \{(\top,p)(b)\}$, whose complement w.r.t $\mathbf{B}(\ulcorner\mathbf{P_3}\urcorner)$ is $M_3 = \{(\bot,p)(a)\}$. The *stratified model* $\mathbf{M}(\mathbf{P})$ of P is the uncoding of M_3, i.e, $\{r(a),s(b),t(a),q(a),q(b),p(b)\}$.

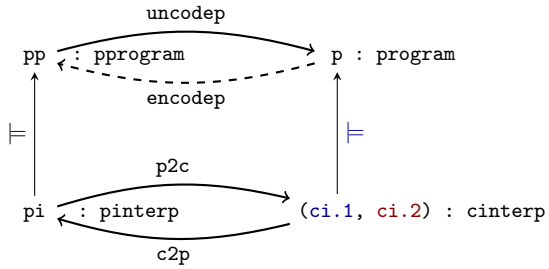
[19] "Positive" interpretations are sets of ground atoms with a `true` flag.
[20] This corresponds to the set of all "positive" ground atoms.
[21] This is the top element of `interp` cf. Section 3.1

4.3 Formal Characterization of the Stratified Evaluation Engine

We first state the main intermediate results, leading up to the key Theorem 16. We assume `p` to be a program; `pp`, `pp1` and `pp2`, "positive" programs; `ci`, an initial complemented interpretation and `pdef`, the default "positive" constant.

Theorem 8 (Complementation Preserves Satisfiability). *If symbols of a stratum* `ss` *do not appear negated in the body of* `p` *clauses, then the satisfiability of* `pp` *w.r.t* `(c2p ci)` *is preserved when complementing* `ci` *w.r.t* `ss`.

Theorem 9 (Encoding/Uncoding Preserves Satisfiability). *In the following, assume* `ci` *is well-complemented. If* `ci.1` *is a model of* `p` *and all* `p` *symbols are in* `ss`*, then* `(c2p ci)` *is a model of* $\ulcorner$`p`$\urcorner$*. If* `(c2p ci)` *is a model of* `pp` *and all* `pp` *body symbols are in* `ss`*, then* `ci.1` *is a model of* $\llcorner$`pp`$\lrcorner$*.*
Intuitively, this is captured by the relations in the informal diagram below:[22]



Theorem 10 (Preservation Properties). *If* `pp` *is safe, its* `pengine_step` *evaluation w.r.t* `ci` *is* sound, bound *by its Herbrand Base,* increasing *and* stable.

Theorem 11 (Symbol Stratifiability). *The atoms outputted by* `pengine_step` *are either in* `ci` *or have symbols appearing in the* head *of* `pp` *clauses.*

Theorem 12 (Positivity). *The "negative" component of* `ci`*, i.e,* `ci.2`*, is not modified by* `pengine_step`*, i.e,* `(pengine_step pdef pp ci).2 = ci.2`.

Theorem 13 (Injectivity). *If* `pp1` *and* `pp2` *are* extensionally *equal, their corresponding* `pengine_step` *evaluations w.r.t* `ci` *are equal.*

Theorem 14 (Modularity). *If* `pp1` *is* safe *and does not contain head symbols in* `pp2` *and* `pi` *is a model of* `pp1`*, then evaluating the concatenation of* `pp1` *and* `pp2` *w.r.t* `pi` *equals the union of their respective evaluations w.r.t* `pi`.

Theorem 15 (Incrementality). *Let* p *be a stratifiable program;* (`ci`, $\mathrm{str}_{\leq}$)*, a cumulative interpretation of* $\mathrm{p}_{\mathrm{str}_{\leq}}$*, and* `ss`*, a stratum. Assume that: 1)* $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq}} \urcorner$ *symbols are not head symbols in* $\ulcorner \mathrm{p}_{\mathrm{ss}} \urcorner$*, 2)* $\mathrm{p}_{\mathrm{str}_{\leq}}$ *symbols are in* $\mathrm{str}_{\leq}$*, 3)* `ci` *is* well-complemented *w.r.t* $\mathrm{str}_{\leq}$*, and 4)* `ci.1` *is a model of* $\mathrm{p}_{\mathrm{str}_{\leq}}$*. The* `pengine_step` *evaluation of* $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$ increments `ci.1` *with facts having symbols in* `ss`.

[22] The dashed `encodep` arrow marks the *partiality* of the cancelation lemma.

Stratified Evaluation Invariant Let p be a stratifiable program and (`ci`, $\text{str}_{\leq}$), a cumulative interpretation of $\text{p}_{\text{str}_{\leq}}$. The *invariant* of *stratified evaluation* `si_invariant` states: 1) `ci.1` is a model of $\text{p}_{\text{str}_{\leq}}$, 2) $\text{p}_{\text{str}_{\leq}}$ symbols are in $\text{str}_{\leq}$, 3) `ci` is *well-complemented* with respect to $\text{str}_{\leq}$, and 4) `ci` symbols are in $\text{str}_{\leq}$.

Theorem 16 (Stratified Evaluation Soundness and Completeness). *Let* p *be a program,* str*, a strata - consisting of lower and upper strata,* $\text{str}_{\leq}$ *and* $\text{str}_{>}$[23] *- and* `ci`*, a complemented interpretation. If the input* cumulative interpretation (`ci`, $\text{str}_{\leq}$) *satisfies the above invariant conditions, then the output* interpretation *of the* one-step evaluation *of* $\text{p}_{\text{str}_{>}}$ *also satisfies them.*
Proof by induction on $\text{str}_{>}$.

As a corollary of Theorem 16, the encoded evaluation engine computes a *model* for a stratifiable program p. A more subtle discussion concerns its *minimality*:

Example 4. Let $P = \left\{ \begin{array}{l} p \leftarrow q. \\ r \leftarrow \neg q. \\ s \leftarrow \neg q. \\ t \leftarrow \neg q. \end{array} \right\} = P_1 \sqcup P_2, P_1 = \left\{ p \leftarrow q. \right\}, P_2 = \left\{ \begin{array}{l} r \leftarrow \neg q. \\ s \leftarrow \neg q. \\ t \leftarrow \neg q. \end{array} \right\}.$
As $M_1 = T_{P_1} \uparrow \omega(\emptyset) = \emptyset$, $M_2 = T_{P_2} \uparrow \omega(M_1) = \{r, s, t\}$, the *computed model* $M_P = \{r, s, t\}$ differs from the *minimal model* $M_P^{min} = \{p\}$.

This is because the minimality of a computed stratified model depends on *fixing* its input. Hence, a model is minimal w.r.t others, if they *agree on the submodel relative to the accumulated stratification.* Since we need to consider the previous and current candidates, we state the `is_min_str_rec` condition independently from the strata invariant conditions. Its proof also follows by induction on $\text{str}_{>}$.

5 Related Work

The work of [20] provides a Coq formalization of the correctness and equivalence of forward and backward, top-down and bottom-up semantics, based on a higher-order abstract syntax for Prolog. Related to our work, as it provides formal soundness proofs regarding the fixpoint semantics, it nonetheless differs in perspective and methodology. Also, while we do not support function symbols and other evaluation heuristics, we do support negation and manage to establish correctness and completeness for the underlying *algorithms* of bottom-up inference. The work in [10] gives a Coq mechanization of standard Datalog in the context of expressing distributed security policies[24]. The development contains the encoding of the language, of bottom-up evaluation and decidability proofs. In our corresponding formalizations, we did not need to explicitly prove the latter, as we set up our types as finite. While we did not take into account modelling security policies, the scope of our established results is wider.

[23] i.e, $\text{str}_{\leq}$ stratifies $\text{p}_{\text{str}_{\leq}}$ and $\text{str}_{>}$ stratifies $\text{p}_{\text{str}_{>}}$
[24] `http://www.cs.nott.ac.uk/types06/slides/NathanWhitehead.pdf`

6 Conclusion, Lessons and Perspectives

The exercise of formalizing database aspects has been an edifying experience. It helped clarify both the fundamentals underlying theoretical results and the proof-engineering implications of making these machine readable and user reusable.

On the database side, it quickly became apparent that, while foundational theorems appeared intuitively clear, if not obvious, understanding their *rigorous* justification required deeper reasoning. Resorting to standard references (even comprehensive ones, such as [2]), led at times to the realization that low-level details were either glanced over or left to the reader. For instance, to the best of our knowledge, no *scrupulous* proofs exist for the results we established. Indeed, as these are theoretically uncontroversial, their proofs are largely taken for granted and, understandably so, as they ultimately target database practitioners. Hence, these are mostly *assumed* in textbook presentations or when discussing further language extensions. It was only by mechanizing these proofs "from the ground up", in a proof assistant, that the relevance of various properties (e.g, safety and finiteness), the motivation behind certain definitions (e.g, predicate intensionality/extensionality, strata restrictions, logical consequence, stratified evaluation), or the precise meaning of ad-hoc notions/notations (e.g, "substitution compatibility", $\mathbf{B}(P)$, model restrictions) became apparent.

As it is well known, database theory is based on solid mathematical foundations, from model theory to algebra. This suggests that, when compared to off-the-shelf program verification, verification in the database context requires that proof systems have good support for mathematics. It was an interesting to discover, in practice, the extent to which database theory proofs could be recast into mathematical ones. To exemplify, by expressing forward chain as an elegant set construct, we transferred proofs about Datalog inference engines into set-theory ones, which are more natural to manipulate. Conversely, when formalizing the stratified semantics of Datalog with negation, we were compelled to resort to some ad-hoc solutions to handle the lack of native library support for lattice theory. Indeed, textbooks largely omit explainations as to why and how it is necessary to reason about such structures when proving properties of stratified evaluation. To this end, we were led to introduce specialized notions, such as interpretation complementation. Also, we had to *explicitly* establish that, at each evaluation step, the Herbrand Base of the program's restriction w.r.t the set of already processed strata symbols was a *well-complemented* lattice.

On the theorem proving side, a crucial lesson is the importance of relying on infrastructure that is well-tailored to the nature of the development. This emerged as *essential* while working on the formalization of standard Datalog. The triggering realization was that, as we could, without loss of generality, restrict ourselves to the active domain, models could be reduced to the finite setting and atoms could be framed as finite types. Therefore, the Mathematical Components library, prominently used in carrying out finite model theory proofs, stood out as best suited for our purposes. Indeed, since we could heavily rely on the convenient properties of finite types and on already established set theory properties, proofs were rendered much easier and more compact.

Apart from having good library support, making adequate type encoding choices proved essential. Having experimented with many alternatives, we noticed first-hand the dramatic effect this could have on the size and complexity of proofs. For example, while having too many primitives is undesirable in programming language design, it turned out to be beneficial to opt for greater base granularity. Separating the type of ground and non-ground constructs helped both at a conceptual level, in understanding the relevance of standard range restrictions, and at a practical one, in facilitating proof advancement. Another example concerns the mechanization of substitutions. Having the option to representing them as finitely supported functions, together with all the useful properties this type has, was instrumental to finding a suitable phrasing for the soundness and completeness of the matching algorithm. Indeed, as the algorithm incrementally constructs groundings, it seemed natural to define an ordering on substitutions leading up to these. Being able to have a type encoding allowing to regard substitutions both as functions and as lists was essential for this purpose. A final example regards the formalization of models. As previously mentioned, setting up the type of ground atoms as finite payed off in that we could use many results and properties from the `fintype` library, when reasoning about models - which was often the case. In particular, we took advantage of the inherent lattice structure of such types and did not need to *explicitly* construct $\mathbf{B}(P)$.

Finally, relying on characteristic properties (the SSReflect P-lemmas), many of which are conveniently stated as reflection lemmas, led to leaner proofs by compositionality. In cases in which induction would have been the default approach, these provided a shorter alternative (also, see [13], which gives a comprehensive fomalization of linear algebra *without induction*).

References

1. *Datalog in Academia and Industry*, Datalog 2.0. LNCS, vol. 7494. Springer (2012)
2. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*. Addison-Wesley (1995)
3. Apt, K.R., Blair, H.A., Walker, A.: *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc. (1988)
4. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: *Design and Implementation of the LogicBlox System*. In: SIGMOD ACM Proc. of ICMD. pp. 1371–1382 (2015)
5. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: *gMark: Schema-Driven Generation of Graphs and Queries*. IEEE Transactions on Knowledge and Data Engineering (2017)
6. Calì, A., Gottlob, G., Lukasiewicz, T.: *Datalog±;: A Unified Approach to Ontologies and Integrity Constraints*. In: Fagin, R. (ed.) ACM Proc. of ICDT. pp. 14–30 (2009)
7. Ceri, S., Gottlob, G., Tanca, L.: *Logic Programming and Databases*. Springer-Verlag (1990)
8. Chin, B., von Dincklage, D., Ercegovac, V., Hawkins, P., Miller, M.S., Och, F.J., Olston, C., Pereira, F.: *Yedalog: Exploring Knowledge at Scale*. In: LIPIcs Proc. of SNAPL. vol. 32, pp. 63–78 (2015)
9. Datomic, `http://www.datomic.com/`

10. DeTreville, J.: *Binder, a Logic-Based Security Language*. In: IEEE Proc. of the Symposium on Security and Privacy. pp. 105– (2002)
11. Doczkal, C., Smolka, G.: *Completeness and Decidability Results for CTL in Coq*. In: Klein, G., Gamboa, R. (eds.) Proc. of ITP. pp. 226–241 (2014)
12. Exeura, `http://www.exeura.com/`
13. Gonthier, G.: *Point-Free, Set-Free Concrete Linear Algebra*. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Proc. of ITP, pp. 103–118 (2011)
14. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S.L., Mahboubi, A., O'Connor, R., Biha, S.O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: *A Machine-Checked Proof of the Odd Order Theorem*. In: Proc. of ITP. pp. 163–179 (2013)
15. Gonthier, G., Mahboubi, A., Tassi, E.: *A Small Scale Reflection Extension for the Coq system* (2016), `https://hal.inria.fr/inria-00258384`
16. Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: *The Lixto Data Extraction Project: Back and Forth Between Theory and Practice*. In: ACM SIGMOD-SIGACT-SIGART Proc. of PODS. pp. 1–12 (2004)
17. Grumbach, S., Wang, F.: *Netlog, a Rule-Based Language for Distributed Programming*. In: Carro, M., Peña, R. (eds.) Springer Proc. of PADL. pp. 88–103 (2010)
18. Hellerstein, J.M.: *The Declarative Imperative: Experiences and Conjectures in Distributed Logic*. ACM SIGMOD Record J. 39(1), 5–19 (2010)
19. Huang, S.S., Green, T.J., Loo, B.T.: *Datalog and Emerging Applications: An Interactive Tutorial*. In: ACM SIGMOD Proc. of ICMD. pp. 1213–1216 (2011)
20. Kriener, J., King, A., Blazy, S.: *Proofs You Can Believe In. Proving Equivalences Between Prolog Semantics in Coq*. In: Schrijvers, T. (ed.) ACM Proc. of PPDP. pp. 37–48. Madrid, Spain (2013)
21. Libkin, L.: *The Finite Model Theory Toolbox of a Database Theoretician*. In: ACM SIGMOD-SIGACT-SIGART Proc. of PODS. pp. 65–76 (2009)
22. Lloyd, J.W.: *Foundations of Logic Programming*. Springer-Verlag (1987)
23. LogicBlox, `http://www.logicblox.com/`
24. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: *Implementing Declarative Overlays*. In: ACM Proc. of SOSP. pp. 75–90 (2005)
25. Lu, L., Cleary, J.G.: *An Operational Semantics of Starlog*. In: Springer ACM Proc. of PPDP. pp. 293–310 (1999)
26. Luteberget, B., Johansen, C., Feyling, C., Steffen, M.: *Rule-Based Incremental Verification Tools Applied to Railway Designs and Regulations*. In: Proc. of FM. pp. 772–778 (2016)
27. The Coq Development Team: *The Coq Proof Assistant. Reference Manual* (2016), `https://coq.inria.fr/refman/`, Version 8.6
28. Ramakrishnan, R., Ullman, J.D.: *A Survey of Research on Deductive Database Systems*. J. of Logic Programming 23(2), 125–149 (1993)
29. Semmle, `https://semmle.com/`
30. Seo, J., Park, J., Shin, J., Lam, M.S.: *Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis*. vol. 6, pp. 1906–1917 (2013)
31. Tarski, A.: *A Lattice-Theoretical Fixpoint Theorem and its Applications*. Pacific J. of Mathematics 5(2), 285–309 (1955)
32. Van Emden, M.H., Kowalski, R.A.: *The Semantics of Predicate Logic as a Programming Language*. J. ACM 23(4), 733–742 (1976)
33. Vardi, M.Y.: *The Complexity of Relational Query Languages*. In: ACM Proc. of STOC. pp. 137–146 (1982)
34. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: *Using Datalog with Binary Decision Diagrams for Program Analysis*. In: Springer Proc. of APLAS. pp. 97–118 (2005)