Developing and certifying Datalog optimizations in Coq/MathComp

Pierre-Léo Bégay
Orange Labs
Lannion, France
Univ. Grenoble Alpes
CNRS, Grenoble INP, VERIMAG
Grenoble, France
pierreleo.begay@orange.com

Pierre Crégut
Orange Labs
Lannion, France
pierre.cregut@orange.com

Jean-François Monin
Univ. Grenoble Alpes
CNRS, Grenoble INP, VERIMAG
Grenoble, France
jean-francois.monin@univ-grenoble-alpes.fr

Abstract

We introduce a static analysis and two program transformations for Datalog to circumvent performance issues that arise with the implementation of primitive predicates, notably in the framework of a large scale telecommunication application. To this effect, we introduce a new trace semantics for Datalog with a verified mechanization. This work can be seen as both a first step and a proof of concept for the creation of a full-blown library of verified Datalog optimizations, on top of an existing Coq/MathComp formalization of Datalog[5, 14] towards the development of a realistic environment for certified data-centric applications.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Theory of computation** → *Program analysis*; • **Networks** → Network dynamics.

Keywords: Datalog, Coq, MathComp, semantics.

ACM Reference Format:
Pierre-Léo Bégay, Pierre Crégut, and Jean-François Monin. 2021. Developing and certifying Datalog optimizations in Coq/MathComp. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439913

1 Introduction

Datalog is a simple and declarative language tuned to data-centric applications. As a first approximation, it is a fragment of Prolog without function symbols. A Datalog program consists of facts, i.e. positive ground atoms, and implicitly universally quantified rules which allow the derivation of new facts. Recursivity makes it possible to compute transitive closures, e.g. accessibility in graphs, in a simpler and more complete way than other query languages, such as SQL, XPath and SPARQL. In contrast to Prolog, the evaluation mechanism of Datalog follows a bottom-up strategy which guarantees termination [2] – in this framework, the set of derivable facts is always finite.

Originally designed as a powerful query language on databases, it has recently gained interest thanks to domain-specific extensions [2, 29]. The introduction of [5] provides a comprehensive list of languages built upon Datalog [3, 9, 16, 27] and applications, in both academic [13, 18] and industrial [11, 15] settings.

This work originates in a large-scale application of Datalog to a telecommunication verification tool aiming at computing connectivity properties in virtual infrastructure managers such as OpenStack. Datalog can model traffic forwarding in various networking elements in only a few lines of code.

Lopes et al have shown in [25, 26] that the Datalog engine must use specific representations of set of values rather than enumeration to efficiently handle common operations on network addresses. They introduced such a structure, called cubes, and the Network Optimized Datalog engine (NoD), that scales to models of networks of industrial size. Although a step in the right direction, NoD imposes constraints on programs to scale. Efficient NoD programs (e.g. [24]) are usually long and complex. The transformation process to generate those programs from the generic ones is manual, error prone, undocumented, difficult to understand and trust by a third-party, and ultimately difficult to maintain when the initial program evolves.

To address these performance and trust issues, we develop two Datalog-level program transformations aimed at speeding-up execution by the NoD engine, as well as a static analysis upon which the first transformation relies. We use and extend the Coq/MathComp formalization of Datalog developed in [5, 14], to show that the static analysis captures an overapproximation of the behavior of a Datalog program, and that the two rewritings preserve the semantics of the

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

CPP '21, January 18–19, 2021, Copenhagen, Denmark

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439913

transformed program. These proofs lead us to design a new Datalog trace semantics, whose implementation is also verified in Coq. The resulting code, which is made available at [1], can be seen as a first and realistic experiment of the aforementioned Coq implementation of Datalog.

Section 2 recalls the basics of Datalog and discusses the limitations of the NoD engine. Sections 3 and 4 present and justify our program transformations and the static analysis. Section 5 introduces and discusses their Coq formalizations and justifications. We sketch out in Section 6 a more efficient version of the static analysis, as well as the main difficulty it raises. Section 7 explains the effects of those optimizations, in particular in our use case. We finally discuss related works in Section 8 and conclude in 9.

2 Datalog

We first present the formal syntax and semantics of Datalog. We then introduce the Network Optimized Datalog engine, and in particular discuss a caveat with the implementation of certain predicates.

2.1 Syntax

Assuming sets $\mathcal{V}$, $\mathcal{C}$ and $\mathcal{P}$ of variables, constants and predicate symbols, programs are built using the following rules:

$$\begin{array}{llll} \mathit{Programs} & P & ::= & C_0, \cdots, C_k \\ \mathit{Clauses} & C & ::= & A_0 :- A_1, \cdots, A_m. \\ \mathit{Atoms} & A & ::= & p(t_0, \cdots, t_{n-1}) \\ \mathit{Terms} & t & ::= & x \in \mathcal{V} \mid c \in \mathcal{C} \end{array}$$

A_0 is the head of the clause, whereas the $A_1 ... A_m$ sequence is its body. In the third rule, $p \in \mathcal{P}$ and n is the arity of p, written $ar(p)$. We will write as $head : clause \rightarrow atom$ and $body : clause \rightarrow 2^{atom}$ the functions that return the head and body of a clause.

Example 2.1. Figure 1 shows a Datalog program fragment, which computes connectivity in a graph (the *linked* predicate) as the transitive closure of the *edge* relation.

$linked(X, Y) :- edge(X, Y).$
$linked(X, Y) :- linked(X, Z), edge(Z, Y).$

Figure 1. Graph connectivity in Datalog

A program contains a mix of ground bodyless clauses, called facts, and rules, i.e. clauses with at least one atom in their body. A predicate is defined only by rules (intensional predicates) or by facts (extensional predicates). In Example 2.1, *linked* is intensional, whereas *edge* (not shown) is considered as extensional, the facts would then represent the set of edges in the graph.

Another restriction, called safety, requires all variables in the head of a rule to appear in its body, ensuring that only a finite number of new facts can be deduced.

Term body occurrences, or *tocc* are 3-tuples in $\mathbb{N}^3$. The components are the indexes of, respectively, the rule, the atom (within the body of the rule) and the argument (within the atom), starting at 0. In Example 2.1, the *tocc* for the Z within *edge* in the second rule would be $\langle 1, 1, 0 \rangle$.

2.2 Semantics

$\mathbf{B}(P)$ is the Herbrand Base of program P, i.e. the set of ground atoms built from its predicates and constants. An *interpretation* I is a subset of $\mathbf{B}(P)$. A substitution ν, i.e. a mapping from variables appearing in the program to (program) constants, can naturally be lifted to clauses. We denote the set of substitutions as Σ.

A clause C is satisfied by an interpretation I if, for any substitution ν, $body(\nu(C)) \subseteq I \Rightarrow head(\nu(C)) \in I$. Lifting this notion to full programs, I is a *model* of P iff all its clauses are satisfied by I. The semantics of P is its unique minimal model w.r.t. set inclusion [2], written $\mathbf{M_P}$. However, this model-theoretic semantics provides no clue on how to actually build $\mathbf{M_P}$. To do so, the following functions are used:

Definition 2.2. (Match condition) A substitution ν matches a clause C w.r.t. an interpretation I, written $match(\nu, C, I)$, iff ν maps all atoms from the body of C to elements of I, i.e. $body(\nu(C)) \subseteq I$.

Definition 2.3. (T_P – Consequence operator) Let P be a program and I an interpretation. The T_P operator adds the set of program consequences to I:
$T_P(I) = \{head(\nu(C)) \mid \nu \in \Sigma \wedge C \in P \wedge match(\nu, C, I)\} \cup I$

Definition 2.4. (Fixpoint evaluation) The iterations of the T_P operator on $\emptyset$ are:

$$\begin{cases} T_P \uparrow 0 = \emptyset \\ T_P \uparrow n+1 = T_P(T_P \uparrow n) \end{cases}$$

Since T_P is monotonic and bound by $\mathbf{B}(P)$, the Knaster-Tarski theorem [34] ensures the existence of a least fixed point, i.e. $\exists \omega, T_P \uparrow \omega = \bigcup_{n \geq 0} T_P \uparrow n$. The fixpoint evaluation of P is defined as $T_P \uparrow \omega = \mathrm{lfp}\,(T_P)$. It is shown in [36] that $\mathrm{lfp}\,(T_P) = \mathbf{M_P}$, ensuring that T_P is an adequate mechanization of Datalog model-theoretic semantics.

Example 2.5. Let P be the graph connectivity program from Example 2.1 augmented with the facts $F = \{edge(1,3), edge(2,1), edge(4,2), edge(2,4)\}$, then
$T_P \uparrow 0 = \emptyset$; $T_P \uparrow 1 = F$
$T_P \uparrow 2 = T_P \uparrow 1 \cup \{linked(1,3), linked(2,1), linked(4,2), linked(2,4)\}$;
$T_P \uparrow 3 = T_P \uparrow 2 \cup \{linked(2,3), linked(4,1), linked(4,4), linked(2,2)\}$;

$T_P \uparrow 4 = T_P \uparrow 3 \cup \{linked(4,3)\} = T_P \uparrow 5$;
The minimal model of P is $\mathbf{M_P} = lfp\ (T_P) = T_P \uparrow 4$.

The previous definitions form the usual definition of Datalog. However, the Coq modelization of [5, 14] separates the rules from the facts, i.e. clauses with an empty body. In the rest of this paper, we also adopt this setting, which matches the philosophy of our trace semantics and static analysis.

Concretely, a "program" will refer to a set of rules, such as Figure 1. A program will be evaluated w.r.t. an initial interpretation, called Extensional DataBase (EDB). This set of initial facts will appear explicitely in the iterations of T_P, now written $(T_P \uparrow n)(I)$, where I is the EDB. The base case is changed to be the EDB, i.e. $(T_P \uparrow 0)(I) = I$, whereas the recursive case is unchanged. This alternative presentation is equivalent to the traditional one, and simply circumvents the first iteration of T_P.

Finally, Datalog engines may introduce **primitive predicates**, i.e. predicates which are not defined *via* facts or rules, but computed on the fly. A classical example of primitive predicate is equality, which may often be useful in practice, and much easier to dynamically check rather than define as the set of all pairs $\langle x, x\rangle$.

2.3 NoD engine and Differences of Cubes

Network Optimized Datalog [25] is a Datalog engine used to check reachability properties of dynamic networks. It is based on a Z3 [28] implementation of Datalog, called μZ [17], with network-oriented modifications made for scalability. One of the key modifications is the use of a data structure called *difference of cubes* for the representation of network packets and their rewriting.

Definition 2.6. (Difference of Cubes) A DoC is a set of patterns *modulo* exceptions. Concretely, they are of the form

$$\bigcup_i (v_i \setminus \bigcup_j v_{i,j})$$

where v_i and v_j are bit vectors patterns with some bits left unspecified. In the setting of NoD, it can be understood as, for example, "every packet of the form v_1 except those matching $v_{1,1}$ or $v_{1,2}$, as well as the packets of the form v_2 with the exception of those matching $v_{2,1}$".

However, in order to scale, NoD programs have to be tailored to reduce the number of interactions between variables from different origin. For example, the contents of the routing tables are inlined in the program rather than kept in a separate base of fact refered by a small generic program. Using NoD at a real-world, industrial level then requires a lot of expertise on NoD itself. Such specialized programs are more complicated to write, understand and above all maintain.

2.4 Handling more genericity

We developed a tool which aims at providing a higher-level network veriication model, using NoD as a backend. Roughly, the tool is used to define generic properties and behaviors as Datalog rules, where the configuration is abstract. Then, the specificities of the analyzed network (routing tables, firewalls, etc) are collected by the tool and used as the EDB.

Example 2.7. The building block of the specification and verification of network behavior is route selection. Figure 2 shows an abstract definition of routing, where `linked(X, Y, P)` states that a packet `P` flow from `X` to `Y`, and `route(X,Y,F,M)` is an extensional predicate describing a route from `X` to `Y` for packets matching `F` (captured by `match_route(P,F)`) with priority `M`. Packets, filters and priority can be arbitrarily complex.

```
linked(X, Y, P) :-
  route(X, Y, F, M), match_route(P, F),
  not exists_better_route(X, Y, P, M).

exists_better_route(X, Y, P, M) :-
  route(X, Y, F2, M2), match_route(P, F2),
  M2 > M.
```

Figure 2. Simplified network reachability in Datalog

In comparison, NoD has a clause for each routing rule, which explicitely contains the negation of the matching of higher-priority rules. Although highly beneficial, this overall *lift* in abstraction and clarity on top of NoD comes at a cost. As illustrated by Example 2.7, it requires the use of primitives, here <, with multiple variables. However, the complexity with which DoC handles some primitives, including all comparisons, changes dramatically with the number of involved variables.

Example 2.8. Figure 3 illustrates how the DoC representation of the $v_1 \geq v_2$ relation is exponential in the number of bits the integers are coded on – here, four.

$$\underbrace{****}_{v_1}\underbrace{****}_{v_2} \setminus \{\underbrace{0***}_{v_1}\underbrace{1***}_{v_2},\ \underbrace{00**}_{v_1}\underbrace{01**}_{v_2},\ \underbrace{10**}_{v_1}\underbrace{11**}_{v_2},$$
$$\underbrace{000*}_{v_1}\underbrace{001*}_{v_2},\ \underbrace{010*}_{v_1}\underbrace{011*}_{v_2},\ \underbrace{100*}_{v_1}\underbrace{101*}_{v_2},\ \underbrace{110*}_{v_1}\underbrace{111*}_{v_2},\dots\}$$

Figure 3. (Incomplete) DoC encoding of the $v_1 \geq v_2$ relation

Example 2.9. Figure 4 illustrates how the comparison between a variable and a constant, here 1101, is linear in the number of bits the integers are coded on.

$$* * * * \setminus \{0 * **, 10 * *, 1100\}$$

Figure 4. DoC encoding of $v \geq 1101$

This observation on the impact on the number of variables in some primitives, combined with the fact that, in practice, the number of relevant integer values (ip addresses, priorities, etc) in networks is much, much lower than allowed by the types, led us to the optimizations which are presented and discussed in the rest of this paper.

3 Partial rule instantiation

This section introduces a first program transformation that aims at the reduction of the number of variables. It consists of a rewriting that replaces a clause with many variables by an equivalent set of specialized versions, which all contain strictly less variables, and a static analysis that provides values for the specialization. We first introduce the intuitions behind the two operations, then formalize the rewriting. The formalization of the static analysis is relegated to Section 5.

3.1 Intuition

In Definition 2.3, the T_P operator *tries out* every possible substitution and, when a match occurs, adds the corresponding fact to the returned set. However, in practice, Datalog engines may try to be more efficient than that. In particular, Datalogcert [6] *builds* the minimal set of relevant substitutions (this point is discussed in Section 5.3).

We observe that an engine which overapproximates the set of candidate substitutions, while still checking the ground atoms in the tail of the instantiated rule against the given interpretation, does not change the deduced set of facts. This means that, if we obtain a projection of such an overapproximation on some variables, we can use it to constrain the tested values. Our first transformation does so by rewriting the program, which avoids the complex and specific operation of changing the internals of the used Datalog engine.

$$\begin{array}{lll} s(X_1, Y_1) & :- & q(X_1), p(X_1, Y_1)\,. \\ q(X_2) & :- & r(X_2, X_2)\,. \\ q(X_3) & :- & f_1(Y_3, Y_3, Z_3)\,,\;\; f_2(X_3, Y_3, Z_3)\,. \\ r(X_4, Y_4) & :- & f_1(X_4, Y_4, X_4)\,,\;\; f_2(Z_4, X_4, Z_4) \end{array}$$

Figure 5. Defining s(X,Y)

Example 3.1. Let us consider the program in Figure 5, where s, q and r are intensional predicates, f_1 and f_2 extensional, and p primitive, as well as the EDB $\{f_2(1, 2, 5), f_2(3, 7, 1), f_1(2, 2, 2)\}$. Without running the program or knowing the behavior of p, one may quickly convince herself that the values of X_1 in practice are a subset of $\{1, 2, 3\}$. With that in mind, we can rewrite the program with the first rule replaced by the three in Figure 6.

$$\begin{array}{lll} s(1, Y_1) & :- & q(1), p(1, Y_1)\,. \\ s(2, Y_1) & :- & q(2), p(2, Y_1)\,. \\ s(3, Y_1) & :- & q(3), p(3, Y_1)\,. \end{array}$$

Figure 6. Defining s(X) with fewer variables

The program indeed has the same semantics, as any computation of the original program is captured by one of the three rules. Conversely, a rule that does not match any concrete computation can not be used, because at least one atom in its instantiated body will not be deduced and available.

Given a way to approximate the values taken by any given variable (or set of variables) of a program, this general method can be used to reduce the number of variables in primitive predicate instances, improving performance in some settings, such as NoD. Before formalizing this transformation, we introduce a static analysis that computes such value sets.

3.2 A static analysis for Datalog

Without loss of generality, we only consider programs where variable names are not shared across rules. We also assume the absence of constants in the head of rules, which can be enforced by introducing new, unary predicates and corresponding facts (see the proof of Theorem 12.5.2 in [2]). We write $p.i$ the ith index (starting at 0) of predicate p.

To convey the idea of our static analysis, we introduce an informal and threefold specification of T_P, illustrated using the program of Figure 5: (1) Given any EDB, the set of values with which $q(X_1)$ can be instantiated is a subset of the union of the values *returned* by the second and third rules. (2) Looking at the former, the set of values for X_2 is a subset of the conjunction of those of any of its instances, i.e. $\langle 1, 0, 0\rangle$ and $\langle 1, 0, 1\rangle$, which correspond to $r.0$ and $r.1$. (3) Given a value for X_2, $\langle X_2, X_2\rangle$ must form a valid tuple of arguments for r, i.e. values used to instantiate an atom must be compatible.

Unlike (3), (1) and (2), which are akin to an alternation between disjunctions and conjunctions, deal with variables and *toccs* individually. They then involve much less computation than the entirety of T_P, while hopefully still providing interesting constraints.

We propose an analysis that performs (1) and (2) by building, for any variable to instantiate, a tree with nodes labeled with $\wedge$ and $\vee$ and leaves representing columns in the EDB tables, e.g. "the third argument of predicate p". The tree then represents the way values flow from the EDB to a variable

during the program's execution. The branches are annotated with the index of the corresponding clause (descendants of ∨-nodes) or atom (descendants of ∧-nodes). We will discuss in Section 6 how to use those annotations to simulate a weaker version of (3) on top of the actual analysis.

Example 3.2. The *flow* of X_1 in the program of Figure 5 is shown in Figure 7. The variable has two occurrences in the body of the first rule, in the first and second atom. However, the second atom is a primitive, which is ignored by the static analysis. The root of of the tree then has only one children, annotated with the index of the atom, 0.

The corresponding atom is an occurrence of q, there are then two descendants, the 1 and 2 annotations being the indexes of the two clauses defining q. The use of a ∨-node reflects the fact that the facts about q is the union of those defined by these two rules (see (1) above).

The right descendant *leads* to the X_3 variable, whose only appearance in the body of the rule is as the first argument of an occurrence of the extensional f_2 predicate. The next descendent then is the corresponding leaf.

On the other hand, the left descendant leads to X_2, which has two occurrences in the same (and only) atom of the rule's body. There are then two new descendants, both annotated with 0, the index of the r atom. Here, the use of a ∧-node is justified by (2). The rest of the tree follows the same logic.

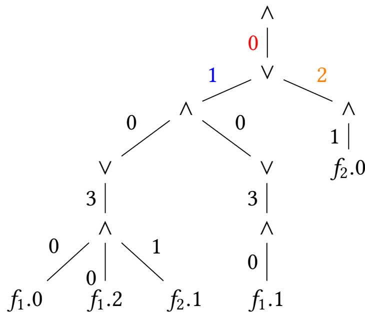


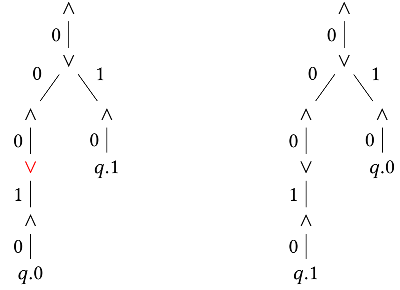
Figure 7. Analysis of a variable

This principle will obviously not fare well with recursion, which happens to be a core feature of Datalog. The analysis then stores all previously visited *tocc*s when recursively calling itself, to avoid having a *tocc* twice in a branch of the returned tree. This bounds the derivations despite potentially recursive programs. The reasoning is that, to find an approximation of the values going through some *tocc*, one should not look at the recursive part of the corresponding predicate, but rather the other predicates that "ground" it. This idea is formalized is 5.2, but let us provide an intuition.

$$p(X_1, Y_1) \ :- \ p(Y_1, X_1)\,.$$
$$p(X_2, Y_2) \ :- \ q(X_2, Y_2)\,.$$

Figure 8. Program P_{rec}

Example 3.3. Figure 9a shows the analysis of variable X_1 in the program of Figure 8, where q is an extensional predicate.



(a) Analysis of X_1 **(b)** Analysis of Y_1

Figure 9. Analysis of a recursive program

The analysis starts with the p atom in the body of the first clause. It then considers two different ways to deduce a p fact, i.e. the two clauses. When looking at the first clause, we analyze the variable matching the previous position, i.e. Y_1. It occurs in the same atom, which can again be instantiated using both clauses. However, this time we have only one child under the ∨-node, as looking at the first clause again would bring us back to our original *tocc*. The analysis eventually captures the fact that values of p can be permuted, and that the set of values of X_1 is a subset those of $q.0$ and $q.1$.

From such a tree, we can extract a set of values for the analyzed variable. Each $f.i$ leaf returns the set of constants appearing at the ith position of a f fact in the EDB, whereas the ∨ and ∧ nodes are treated as ∪ and ∩, respectively.

3.3 Clause duplication formalization

We now fully define the transformation that was informally introduced in Section 3.1. It first assumes a program P and an interpretation I. As one may want to only instantiate a selection of variables, it is parameterized by such a subset R. It also requires a set of substitutions S that captures the behavior of the program projected on the variables in R.

To formalize this notion, we first introduce $\nu_{|R}$, the restriction of ν to the variables in R, as well as *vars*, the function that computes the set of variables in a clause. A first clarification of the completeness condition of S is that, whenever a substitution ν matches a clause C at some point of the program's execution, then the projection of ν over the relevant variables is in S.

By relevant variables, we mean the intersection of the variables in the C clause and the set of variables to instantiate R. Moreover, due to the monotonicity of the matching w.r.t. the provided interpretation, a substitution matches a clause at some point of the execution if and only if it matches it w.r.t. the semantics of the program, i.e. $(T_P \uparrow \omega)(I)$. The resulting formula is as follows:

$$\forall C \in P, vars(C) \cap R \neq \emptyset \\ \Rightarrow (\forall v, match(v, C, (T_P \uparrow \omega)(I)) \Rightarrow v_{|R} \in S).$$

It might seem paradoxical to mention the full semantics of a program to introduce an optimization. We then want to clarify this point, by underlying that the completeness condition mentioned above has to be checked once for any static analysis which computes a set of candidate substitutions, such as seen in 3.2, and plays no role in practice.

Using function *dom*, that returns the domain of a substitution, any clause can be instantiated using the *inst* function, which can naturally be lifted to programs.

$$inst(C) = \begin{cases} \{v(C) \mid v \in S \ \wedge \ dom(v) = R \cap vars(C)\} \\ \qquad \text{if } vars(C) \cap R \neq \emptyset \\ \{C\} \qquad \text{otherwise} \end{cases}$$

Note that, in these definitions, the $vars(C) \cap R \neq \emptyset$ condition can be dropped if the empty substitution is manually added to S.

We can now use the T_P operator to state and prove that the two programs deduce the same facts in the same number of steps. The semantic adequacy trivially follows.

Theorem 3.4. ***(Transformation completeness)*** *For any number of steps k, $(T_P \uparrow k)(I) \subseteq (T_{inst(P)} \uparrow k)(I)$.*

Proof. We proceed by induction on k. In the base case, we have $(T_P \uparrow 0)(I) = (T_{inst(P)} \uparrow 0)(I) = I$.

In the recursive case, let $f \in (T_P \uparrow k+1)(I)$. We can extract a clause C from P and a substitution v such that v matches C w.r.t. $(T_P \uparrow k)(I)$ and f is the head of $v(C)$. If C has no relevant variable, i.e. $|vars(C) \cap R| = 0$, we reuse the same clause and substitution. Since the atoms in the body of the instantiated clause are in $(T_{inst(P)} \uparrow k)(I)$ (induction hypothesis), meaning that we can indeed deduce f in $inst(P)$.

Otherwise, we use the completeness hypothesis on S to show that $inst(P)$ contains $v_{|R}(C)$. We write as $v_{|\overline{R}}$ the restriction of v to the variables not in R, we also show that, using $(T_{inst(P)} \uparrow k)(I)$ as an interpretation, this partially instantiated clause matches $v_{|\overline{R}}$ to produce f. This is a corollary of a more general lemma, stating that whenever the variables are split into two sets X_1 and X_2, and a substitution σ matches a clause C', then $\sigma_{|X_2}$ matches $\sigma_{|X_1}(C')$. □

Theorem 3.5. ***(Transformation soundness)*** *For any number of steps k, $(T_{inst(P)} \uparrow k)(I) \subseteq (T_P \uparrow k)(I)$.*

Proof. This lemma works in a both similar and dual way, as we get two substitutions (one for the clause instantiation, one matching the transformed clause) that need to be combined to retrieve the substitution used in the original program. □

The reader may notice a discrepancy between the analysis and the rewriting, as the former provides values for only one variable, and the latter assumes a set of substitutions, i.e. potentially instantiates multiple variables at once. We wanted these two components to be kept separate, so that the rewriting can be used with another analysis. In particular, one may come up with a static analysis that returns a set of values for multiple variables at once, which could then be directly used with the rewriting, encapsuling this set of values into substitutions. Such an analysis is sketched and discussed in Section 6.

In practice, after using the analysis on multiple variables of the same rule, one can either apply the rewriting multiple times with a substitution on a single variable, or generate substitutions using a cross product of the different value sets.

4 Predicate specialization

After the number of variables, our second optimization reduces the number of arguments of some predicates, by introducing new symbols to partition existing relations into smaller ones. Given an intensional predicate such that one of its arguments is always a constant in the rules defining it, then the predicate can indeed be replaced by a set of specialized versions.

```
S(1,Y,Z)  :-  Q(Y,Z).        |    S1(Y,Z)  :-  Q(Y,Z).
S(2,Y,Z)  :-  R(Z,Z,Y).      |    S2(Y,Z)  :-  R(Z,Z,Y).
T(X)  :-  S(1,X,X).          |    T(X)  :-  S1(X,X).
U(X)  :-  S(X,X,X).          |    U(X)  :-  S(X,X,X)
```

Figure 10. Normal and specialized program

Example 4.1. The first two rules on the left of Figure 10 define a predicate S of arity 3, and the third and fourth rules use it. We introduce the S_1 and S_2 predicates, of arity 2. The rules can then be replaced by those on the right. To allow the use of rules still containing S atoms in their body, such as the fourth one, we need to add the rules of figure 11 to the new program.

```
S(1,Y,Z)  :-  S1(Y,Z).
S(2,Y,Z)  :-  S2(Y,Z).
```

Figure 11. Relating normal and specialized definitions

Note that, on the other hand, the addition of the reverse rules (normal to specialized version, e.g. $S_1(Y, Z) \longleftarrow S(1, Y, Z)$) is not required.

Assuming a Datalog program P where the ith argument of a predicate S is always a constant, and writing P' the program obtained via the specialization of S at index i, we proved the following results:

Theorem 4.2. ***(Predicate specialization completeness)*** *For any number of steps k*, $(T_P \uparrow k)(I) \subseteq (T_{P'} \uparrow 2k)(I)$.

The number of steps used in the transformed program is doubled, because of the use of rules such as those shown in Figure 11. The transformed program produces specialized facts that did not appear in the original program. This is not a concern for the completeness theorem, as the new facts are on the right side (both literally and figuratively), but the soundness must be formulated *modulo* those new facts:

Theorem 4.3. ***(Predicate specialization soundness)*** *For any number of steps k*,
$\{x \in (T_{P'} \uparrow k)(I) \mid x \text{ is not specialized}\} \subseteq (T_P \uparrow k)(I)$.

One might expect that the number of steps used in the transformed program might again be doubled compared to the original one, but that would not account for the normal, not specialized part of the program.

5 Coq formalization

The Coq certification of the static analysis relies on a new Datalog trace semantics we first introduce. We then outline the formalization and verification of the analysis, as well as the two program transformations.

The Coq definitions and statements are slightly altered[1]. Also for space constraints and readability, the hypotheses are removed from the individual lemmas and theorems, but described in Section 5.5. Finally, we discuss some choices and errors made in the course of this work.

5.1 Datalog trace semantics

A trace semantics not only assigns a meaning to a program, but also keeps track of the computations leading up to it [12]. Datalogcert [6] already formalizes the fixpoint semantics of Datalog (see 2.2), but the proof of our static analysis reasons about the full deduction of a fact *as a whole* and *a priori*, in the statement of a core intermediate lemma (see 5.2). We then need to introduce a Datalog trace semantics, formalized as a set of trees and written $\mathbf{B}_T(P)$. The leaves are facts taken from the initial interpretation, while internal nodes represent a deduction via a rule, a substitution (both stored in the node as a couple) and a previously deduced set of facts (the descendants in the tree). We first need a function that maps a trace to the corresponding deduced fact:

Definition 5.1. **(Erasing function from trees to facts)**

$$ded(x) = \begin{cases} f & \text{if } x = Leaf(f) \\ \nu(head(C)) & \text{if } x = Node(\langle C, \nu\rangle, descendants) \end{cases}$$

We adapt the base semantics to its trace version. First, the interpretation I must be a set of trees rather than facts:

Definition 5.2. **(tb – Interpretation to trees)**

$$tb(I) = \{Leaf(f) \mid f \in I\}$$

In the spirit of Section 2.2, the trace semantics is defined *via* an operator called Tt_P. Like T_P, it is iterated to build new traces on top of the previously deduced ones. When deducing a new fact with a clause C and a substitution ν, the previous iteration must contain traces for the body of $\nu(C)$:

Definition 5.3. **(Tt_P – Consequence operator on traces)**

$$\begin{aligned} Tt_P(I_t) = I_t \cup \{&Node(\langle C, \nu\rangle, [F_1, ..., F_n]) \in \mathbf{B}_T(P) \\ &\mid \quad C = A_0 :\!- A_1, \cdots, A_n \in P \\ &\wedge \forall i \in [1..n], F_i \in I_t \ \wedge \ ded(F_i) = \nu(A_i)\} \end{aligned}$$

Example 5.4. Let us consider again the program of Example 2.1, with the first and second clauses denoted C_0 and C_1. The trace semantics contains the four traces shown in Figure 12, which gradually lead to the deduction of $linked(4, 3)$. Writing t_0 to t_3 the trees, $ded(t_0) = edge(4, 2)$, $ded(t_1) = linked(4, 2)$, $ded(t_2) = linked(4, 1)$ and $ded(t_3) = linked(4, 3)$.

Since this is our own semantics, we need to prove its adequacy with the usual one. Given a program P with interpretation I, this result is expressed in the following lemmas, which are both proved by induction on the left deduction:

Lemma 5.5. ***(Datalog trace semantics completeness)*** *For any number of steps k*,
$\forall x \in (T_P \uparrow k)(I), \ \exists t \in (Tt_P \uparrow k)(tb(I)), \ ded(t) = x$

Lemma 5.6. ***(Datalog trace semantics soundness)*** *For any number of steps k*,
$\forall t \in (Tt_P \uparrow k)(tb(I)), \ ded(t) \in (T_P \uparrow k)(I)$.

Unlike $\mathbf{B}(P)$, $\mathbf{B}_T(P)$ is not finite by default. In our Coq formalization, $\mathbf{B}_T(P)$ is defined using the finType `wutree` [4], i.e. the type of trees with a bounded width (here the maximal body length amongst the clauses of a given program) and unicity across paths. The unicity constraint bounds the height of the trees by the cardinal of the node type.

The substitutions were already defined as a finType in [6], and we changed the clause type so that it is finite as well (see Section 5.5). The completeness lemma remains true in this setting, as having a repetition in a deduction amounts

[1]Type names are changed to better match those in the paper, and some meaningless arguments, e.g. default values provided for nth, are removed.

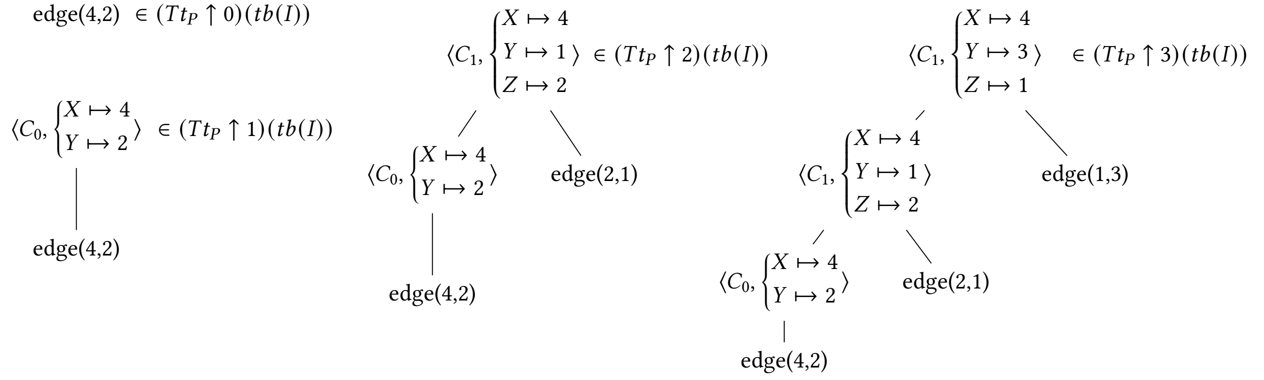


Figure 12. Deduction of $linked(4, 3)$

to proving x inside a proof of x. The whole deduction can be replaced by the inner one, until a repetition-free tree is reached.

5.2 Justification of the static analysis

As stated in Section 3.2, the trees produced by the analysis are akin to propositional formulae. The Coq formalization handles them in Disjunctive Normal Form. In this form, the relation with the actual computation seems less natural, as we lose the alternation between conjunctions and disjunctions, but is surprisingly much easier to prove.

Building a DNF means that we need to be able to compute every combination of propositional formulae across different disjunctions – here, sets. Figure 13 shows our generalization of the cartesian product between two sets provided by MathComp (see `setX` in finset.v), where function `tnth x j` returns the $\mathtt{j}^{th}$ element of tuple x.

```
Definition gen_cart_prod
   {A : finType} (ss : seq {set A})
 : {set (size ss).-tuple A} :=
  let m := size ss in
  [set x : m.-tuple A |
    [forall j : 'I_m, (* seq -> tuple *)
      tnth x j \in tnth (in_tuple ss) j]].
```

Figure 13. Generalized cardinal product

The DNF is encoded as a set of sets of sequences. The *outer* and *inner sets* represent the disjunctive and conjunctive parts, respectively. Figure 14 shows the Coq definition of the static analysis (some auxiliary but fundamentally straightforward functions, such as computing the set of occurrences of a given variable, are omitted). In this definition, `prev` is the set of previously visited *tocc*s, and `v` is the studied variable. The role and value of `count`, as well as the coding style are discussed later on.

Although the general principle of the analysis is rather straightforward, the actual definition is not. Despite its shortness and heavy use of MathComp's useful set comprehension notations, ensuring that it actually computes an overapproximation of Datalog semantics does not seem superfluous.

As previously stated, the analysis does not consider the same program point twice, allowing a fast and terminating analysis of recursive programs. However, the resulting, truncated trees and the actual deduction of facts, i.e. traces, do not fully overlap one another in this setting (which was the case with a previous version of the analysis, see Section 6).

To relate them, we introduce an intermediate layer, which we call the *no-recursion trace*. The idea is to identify a truncated trace that still preserves enough information to be related to the actual trace semantics – a problem already tackled in other verification contexts [19]. More concretely, given a deduction trace and a variable, we extract repetition-free sequences of *tocc*s the variable goes through.

Example 5.7. We reuse the program from Example 2.1 and the rightmost trace of Figure 12, which represents a deduction of $linked(4, 3)$. Let us compute the no-recursion trace of X in the second rule. X has only one occurrence in the body of the rule, at index 0 of the *linked* atom. We then look at the child corresponding to the atom – the left child – of the actual trace, which contains the same clause. The corresponding term, i.e. the one at index 0 of the head of the clause, is again X, which only occurs in the *linked* predicate.

Since this *tocc*, $\langle 1, 0, 0\rangle$, has already been visited, it is not added again to the sequence. In a sense, we ignore this step and keep exploring the trace, which leads us to the first clause. We add the *tocc* of X in the *edge* predicate, i.e. $\langle 0, 0, 0\rangle$. The next child in the trace is a leaf (the *edge* predicate is extensional), so we stop here and return the set only containing the sequence $[\langle 1, 0, 0\rangle, \langle 0, 0, 0\rangle]$. This sequence indeed is a truncated, repetition-free version of the *path* taken by values from the EDB to the variable w.r.t. the actual trace.

```
(* flattening a cartesian product *)
Definition bigcup_cart {m} {A : finType} (s : {set m.-tuple {set A}}) : {set {set A}} :=
  [set \bigcup_(x <- tval y) x | y : m.-tuple {set A} in s].

Fixpoint analyze_var_prev (prev : {set occ}) (v : var) (count : nat) : {set {set (uniq_seq occ)}} :=
  match count with | 0 => set0 | count.+1 =>
    (* occurrences of v not yet visited *)
    let occs := occsInProgram p v :\: prev in
    let analyze_pi (prev : {set occ}) (o : occ) :=
        match p_at o with (* predicate of the atom at the occurrence *)
         | None => set0
         | Some f =>
            match predtype f with
              | Edb => [set [set unil]]
                        (* get_cl_var cl i returns the variable at the i-th position of cl's head     *)
                        (* t_ind returns the third component of an occurrence, i.e. t_ind <x,y,z> = z *)
              | Idb =>  let arec := [set (analyze_var_prev prev (get_cl_var cl (t_ind o))) count
                                      | cl in p & head_predicate cl == f]
                        in \bigcup_(x in arec) x end end in

    (* adding occurrence o on top of every branch in dt *)
    let all_add_o (dt : {set {set (uniq_seq occ)}}) (o : occ) : {set {set (uniq_seq occ)}}:=
      [set [set o::br | br in ct] | ct in dt] in

    let arec := [seq all_add_o (analyze_pi (occ |: prev) occ) occ | occ <- enum occs] in
    bigcup_cart (gen_cart_prod arec) end.

(* Version used in practice, with prev set to empty set for maximal precision *)
Definition analyze_var (v : var) (count : nat) : {set {set (uniq_seq occ)}} :=
  analyze_var_prev set0 v count.
```

Figure 14. The static analysis in Coq

```
Theorem no_rec_needed tr v (i : interp)  cl
                           (m : nat) s :
   tr \in sem_t p m i
-> root tr = inl (RS cl s)
-> v \in tail_vars (body_cl cl)
-> [forall br in norec prev tr v (height tr).+1,
      ...]
```

Figure 15. Completeness of no-recursion traces

The Coq definition of the no-recursion trace also has a *count* argument to ensure its termination. Its *natural value* is the successor of the height of the given trace, as seen in the lemma of Figure 15 (we omit the full details of its Coq formalization for space reasons). This lemma states that, for any deduction which ultimately uses a clause *cl* with a substitution σ, and for any variable v that appears in *cl*, the sequences extracted from the trace all lead to a (potentially different) $\langle f, i \rangle$ such that there exists a fact $f(\overrightarrow{c})$ in the EDB with $\sigma(v) = \overrightarrow{c}.i$.

In other words, the no-recursion trace is used to compute the bounded part of the constraints which the actual semantics enforces. In that sense, it is complete. Another interpretation is that, as stated in Section 3.2 and intuited by the theorem's name, dealing with recursion is not necessary to get a first approximation of the semantics of a program.

Now we need to relate the no-recursion trace with the actual static analysis. With the no-recursion trace being a set of sequences and the analysis encoded as a set of sets of sequences, the core lemma simply states that the no-recursion trace is actually an element of the analysis, meaning that it is captured, as shown in Figure 16.

The program analysis also uses a fuel argument called `count`. Our first intuition for its concrete value was the cardinal of *tocc*, i.e. the number of *tocc*s of the given program. Although it probably is an adequate value, a surprising, different answer emerges from `no_rec_capt`, the lemma relating the no-recursion trace and the analysis.

```
Lemma no_rec_capt prev tr i m cl s v :
   tr \in sem_t p def m i
-> root tr = inl (RS cl s)
-> v \in tail_vars (body_cl cl)
-> norec prev tr v (height tr).+1
 \in analyze_var_prev prev v (height tr).
```

Figure 16. No-recursion trace capture

In this lemma, the fuel of the analysis is the successor of the height of the trace. It is in itself not satisfactory as the deduction traces can in theory be arbitrarily long but, as mentioned in 5.1, we implement the traces as trees with a height bounded by the cardinal of the node type, here couples of a clause and a substitution, written `rul_gr`. Once the monotonicity of the analysis w.r.t. fuel is shown, its normal form can be defined using the product of the cardinals of the clause and substitution types. The corresponding lemma is shown in Figure 17, where `rul_gr` is the node type of the traces.

```
Lemma no_rec_capt prev tr i m cl s v :
   tr \in sem_t p def m i
-> root tr = inl (RS cl s)
-> v \in tail_vars (body_cl cl)
-> norec prev tr v (height tr).+1
 \in analyze_var_prev prev v #|rul_gr|.
```

Figure 17. No-recursion trace capture, normal form

Combining this result with Theorem 15, we show that the analysis can be used to extract a set of substitutions that overapproximates the semantics of the studied program, in the sense formulated at the beginning of Section 3.3.

5.3 Clause duplication

The Coq proofs of theorems 3.4 and 3.5 are fundamentally similar to the paper versions above. Our main technical difficulty for those proofs was the use of substitutions and matchings. The original development [6] of [5] provides a constructive matching function, that returns the set of minimal substitutions matching a clause w.r.t. an interpretation, but surprisingly no boolean one. The "minimality" of the constructive version leads to complex proofs, requiring specific and intricate induction principles. After struggling for a long time with this version (some desired results could be proved, others could not), we defined a function that checks whether a given substitution matches a clause w.r.t. an interpretation, and related both definitions. This boolean matching was also most helpful in the proof of the predicate specialization transformation.

```
Definition dep_iota (m n : nat)
 : seq ('I_(m+n)) :=
    pmap insub (iota m n).

(* [X_1, X_2, ..., X_j] *)
Definition gen_vars_j (j : 'I_n.+1): seq term :=
 map
  (fun x => Var x)
  (map (fun x : 'I_j => widen_ord (ltn_ord j) x)
       (dep_iota 0 j)).
```

Figure 18. Manually defining a sequence of variables

The static analysis and clause duplication are completely separated in our Coq development. Given another method to extract a set of substitutions from a Datalog program, one then only needs to show that it has the completeness property expressed in 3.3 to show that the use of the partial rule instance on top of this method preserves the semantics.

5.4 Predicate specialization

Theorems 4.2 and 4.3 are proved by induction on the number of steps in the deduction. However, formulated as they are above, the provided inductions are not strong enough. We write as $proj$ the function that takes a fact $F(c_1, ..., c_n)$ and returns $S_{c_i}(c_1, ..., c_{i-1}, c_{i+1}, ..., c_n)$ if F is the predicate to be specialized S, and behaves as the identity otherwise. The inductions are performed on the following lemmas, from which theorems 4.2 and 4.3 can easily be deduced:

Lemma 5.8. ***(Predicate specialization completeness)*** *For any number of steps k,*
$(T_P \uparrow k)(I) \cup \{proj(x) \mid x \in (T_P \uparrow k)(I)\} \subseteq (T_{P'} \uparrow 2k)(I)$.

Lemma 5.9. ***(Predicate specialization soundness)*** *For any number of steps k,*
$(T_{P'} \uparrow k)(I) \subseteq (T_P \uparrow k)(I) \cup \{proj(x) \in (T_P \uparrow k)(I)\}$.

The main difficulty in proving those results was the use of the rules as in Figure 11. Indeed, the point of [6] is the verification of a Datalog engine, i.e. ensuring that iterating the T_P operator on any given program will compute the expected semantic. On the other hand, we have to manually add and use such rules. In [6], the atoms carry a proof that their number of arguments is the arity of the associated predicate, and the variables are encoded as ordinals, meaning that we need to deal with a lot of dependent types. As an illustration, Figure 18 shows the (somewhat cumbersome) definition of a sequence of variables used in the generic rules.

To use those generic rules, we wrote a function that takes a list of variables and a list of terms, and creates a substitution that maps each variable to the corresponding value. This function uses a deduced specialized fact to build the

substitution that matches a generic rule. A shift in the list of variables fully changes the extracted substitution, meaning that lemmas on matching using this function could not be proved using straightforward inductions. Another difficulty to certify the predicate specialization was to find well-suited abstractions for the list of variables using a combination of properties and functions.

Defining and certifying this rewriting felt like working *against* [6], which resulted in core lemmas that are much more abstruse than the other results presented in this paper. It could however, along with the definitions and tricks it relies on, be used if other program transformations that add rules are to be introduced and verified in the future.

5.5 Assumptions

The original development [6] assumes finTypes for the predicate names and constants, an arity function, a number of variables, a default constant, and a program with the safety condition defined in Section 2.1. We assume a program-specific maximum length for the bodies of the clauses, and default values (predicate, variable, etc) for the `nth` function. The definition of `wutree` requires a default ground atom.

We added three other hypotheses previously mentioned to the syntax of Datalog: a predicate appears at the head of a clause iff it is intensional (cf. 2.1), variables are not shared across clauses and there is no constant in the heads of clauses (see Section 3 for both). Those last two assumptions could be replaced by actual transformations ensuring the desired properties (i.e. adding rule index to variable names for the former, and a unitary predicate for each constant appearing in a clause's head for the latter), but it was not tackled in the course of this work.

The proofs of the predicate specialization require a few technical hypotheses, such as the number of variables being larger than the arity of the specialized predicate. Finally, the use of `Program.Equality` to certify the trace semantics requires the heterogeneous equality `JMeq_eq`, and [4] uses Equations [33] to build finTypes, which implies the functional extensionality axiom. The full detailed list of assumptions can be found in the README of [1].

5.6 Discussion and lessons learned

The finite nature of Datalog allows the authors of Datalogcert [6] to make the most of MathComp's finTypes and set notations throughout their development. We leverage this asset as well, and although it sometimes implied to rely more on dependent types to fit our structures into finTypes, we believe the trade-off was beneficial, as it led to much more readable, paper-like definitions and statements. The proof effort was also lightened, as these definitions made it easier to follow the high-level proof structures, rather than get lost in the kind technicalities sometimes required to encode concepts in Coq.

Section 6 of [35] states that "generally speaking, there are two ways to specify an algorithm in Coq: either as inductive predicates using inference rules, or as computable functions defined by recursion and pattern-matching over tree-shaped data structures". Datalog programs however, are not naturally seen as trees, both from syntax[2] and semantics standpoints. Our first approach was to develop our static analysis in the form of a typing system, and formalize it in Coq using four mutually-defined inductives. The result, which was supposed to emulate a loop-based implementation of the idea behind the analysis, was not satisfactory, as it lacked clarity, was hard to use in practice (the induction principle required four manually-defined invariants and produced many proof obligations), and did not allow to reason about the termination of the analysis, which we felt was really missing. As a corollary, there was also something awkward about defining what is supposed to be a deterministic function as an inductive predicate.

We eventually switched to the set-based definition seen in Figure 14. Although the definition is not completely straightforward, the intricacy seems inherent to the analysis rather than a consequence of the formalization itself.

The authors of [35] also recall that defining a function, such as our analysis, in a computational way rather than as an inductive also allows its extraction as an Ocaml program (which we have not experimented with this development yet). Alternatively, numerous non-functional programming languages (e.g. Python) now support set notations. Another advantage of this version of the analysis is then its simplified translation in many languages, which reduces the gap between formalization and implementation, and makes the latter more trustworthy.

Although the use of finTypes and set notations was eventually most beneficial to us, our proof style remained more classical. This is in contrast to [6], which uses SSReflect extensively. Combined with the heavy use of dependent types to obtain finTypes, it resulted in a development that was probably longer than what could be expected (approx. 6500 loc, vs. 1500 for [6]).

The conclusion of [5] underlines that the justification of many foundational and "intuitively clear" database results had always been treated with a high-level persective rather than "scrupulous proofs", meaning that "low-level details were either glanced over or left to the reader". This also applies to our present work, in particular the static analysis, where fundamentally simple ideas can be implemented in a rather intricate and error prone way.

[2] Although they implement them as lists in the formalization, Datalog programs are even defined as sets of clauses in [14].

A preliminary version of the analysis, as an inductive, contained a very subtle but major error that made the analysis return a special, theoretically never used *no info* result when applied to recursive programs. Since our lemmas were defined as "if the analysis returns an actual result, it is complete", it went unnoticed at first. We understood there was a problem when reflecting on the proofs, noticing that Datalog recursion was not dealt with. We had not realized that when actually writing the proofs, because of the very technical, sometimes obscure, obligations generated by the four mutually-defined inductives. On the other hand, working with the set-based version was much clearer and allowed easier high-level reasoning.

In summary, even in the context of machine-aided verification, the mix of an error in a minor definition – which would have been spotted with correct lemmas – and lacking formulations of completeness properties – which would have been benign with correct definitions – could lead to a broken result. Computational definitions, higher-level tools (both provided by MathComp in our case) and a more introspective view should help avoid this kind of situation.

The simple ideas behind our static analysis are not only error prone at the level of implementation, but sometimes also conceptually. The next Section introduces a component of the rule instantiation, that seemed like a very natural and efficient addition, and discusses how the verification process helped us realize that it was not adequate for every program.

6 Using dependencies across analyses

In 3.2, we identified three components of the T_P operator and designed a static analysis that performs the first two. However, the two definitions of p in Figure 19, which behave differently, can only be distinguished using the third component, that relates the values assigned to different variables.

$$p(X,Y) \text{ :- } q(X,Y). \quad | \quad p(X,Y) \text{ :- } q(X,Z_1), q(Z_2,Y).$$

Figure 19. Linear and quadratic definitions of p

The first definition computes a number of p-facts that is linear in the number of q-facts, while it is quadratic in the second. In other words, when used on multiple variables of a same rule, the first two components of T_P alone may produce many useless clauses because they fail to track dependencies, such as between X and Y on the left of Figure 19.

Such constructs were present in our use case, sometimes with many dependent variables. Figure 20 presents a simplified implementation of the firewall mechanism (the protocol is not taken into account), where the presence of many primitives, including comparisons, requires to partially instantiate the rule. Every of the 10 variables that appear are *linked* by the `firewall_rule` extensional predicate, meaning that using a cartesian product will lead to a serious impact on performances.

```
ok_fw_rule(RID, IP1, IP2, PRO, P1, P2) :-
  firewall_rule(id=RID,
                protocol=PRO,
                source_prefix=P1,
                dest_prefix=P2,
                source_mask=M1,
                dest_mask=M2,
                src_port_min=MIN1,
                src_port_max=MAX1,
                dest_port_min=MIN2,
                dest_port_max=MAX2),
  P1 = IP1 & M, P2 = IP2 & M,
  P1 >= MIN1, P1 <= MAX1,
  P2 >= MIN2, P2 <= MAX2.
```

Figure 20. Simplified firewall in Octant

Moreover, things can be more obsfuscated than with the previous examples: Figure 21 illustrates how a dependency can be preserved (left side) or lost (right side) across rules.

$$\begin{array}{lll} p(X,Y) \text{ :- } q(X,Y). & | & p(X,Y) \text{ :- } q(X,Y). \\ q(X,Y) \text{ :- } r(X,Y). & | & q(X,Y) \text{ :- } r(X,Z_1), \ r(Z_2,Y). \\ r(X,Y) \text{ :- } s(X,Y). & | & r(X,Y) \text{ :- } s(X,Y) \end{array}$$

Figure 21. Deep linear and quadratic definitions of p

Our analysis was first designed to deal with this issue, which is why the branches of the trees are annotated to store the corresponding atoms or rules. The idea was to overlap the trees resulting from the analyses of multiple variables in a single clause, using the annotations to exclude incompatible flows, e.g. an atom being instantiated with different rules defining the corresponding predicate.

Example 6.1. In Example 3.3, variables X_1 and Y_1 appear in the same clause. The roots of their respective analyses (Figures 9a and 9b) have only one descendant, annotated with 0 in both cases. This implies that the corresponding atom is the same. Then, we have $\vee$-nodes, and two descendants, annotated with 0 and 1. The left (resp. right) subtree represents in both cases a use of the first (resp. second) clause. Mixing the values extracted from the left branch of one of the trees and the right branch of the other tree would amount to simulate a program execution where the p atom in the body of the first clause is instantiated with both the first and second clause at the same time. We can exclude this possiblity.

Figure 22 shows the overlapping of the two analyses using annotations. The result states that the values for X_1 and Y_1 can be extraced from the q facts of the EDB, either directly (left branch) or after permutation (right branch). More formally, X_1 and Y_1 can be instantiated using the values in $\{\langle x, y\rangle \mid q(x, y) \in EDB \vee q(y, x) \in EDB\}$, which matches exactly the actual behavior of the program.

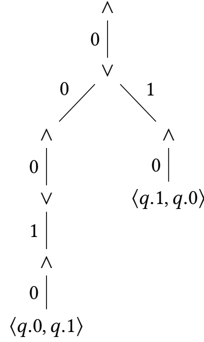


Figure 22. Merged analyses

This example shows the annotations in the trees produced by the analysis can be used to avoid extracting many irrelevant values. Unfortunately, this operation is not safe.

$$p(X_1, Y_1, Z_1) \ :- \ p(Y_1, X_1, Z_1) .$$
$$p(X_2, Y_2, Z_2) \ :- \ q(X_2, Y_2, Z_2)$$

Figure 23. Mixing some values

Example 6.2. Consider the program in figure 23. The analysis returns the trees shown in figure 24. If we merge the trees of Z_1 and X_1 (resp. Y_1), we obtain that X_1 (resp. Y_1) can only be instantiated using values from $q.1$ (resp. $q.0$) rather than both $q.0$ and $q.1$, meaning that the result is not complete.

As explained in Section 5.6, a preliminary implementation of the analysis only considered non-recursive programs. In that setting, the no-recursion trace introduced in Section 5.2 was unnecessary, as we could prove that any trace directly matched a subtree of any analysis. When analyzing multiple variables in a given clause, the returned trees then all contained a subtree that fully matched the same trace. We proved that there is no inconsistency between trees that match the same trace, thus ensuring that excluding incompatible branches did not break the completeness property.

Based on this result, we expect that there is a class of Datalog programs, strictly larger than non-recursive ones, where recursion is only used in a way that allows this more precise version of our analysis. Our first intuition is that a program is *homogeneously recursive* if, for any given rule, all *argument cycles* have the same length. For example, in Figure 23, the cycles of X_1 and Y_1 have length 2, whereas it is 1 for Z_1. This remains to be formalized and verified.

7 Effects of the optimizations

As explained in Section 2.4, the Network Optimized Datalog engine uses a representation called *Differences of Cubes*, which does not fare well with some primitive predicates. The complexity of the computation of these predicates grows exponentially in the number of (bits across) cubes, i.e. variables, in their instances. The goal of our optimizations is then to minimize both the number of variables and sizes of cubes in any given program.

The clause specialization reduces the number of variables occuring in primitive predicates, but also specializes the head of rules that directly depend on facts from the EDB. This allows and fosters the use of the predicate specialization, which reduces the sizes of the cubes used in NoD.

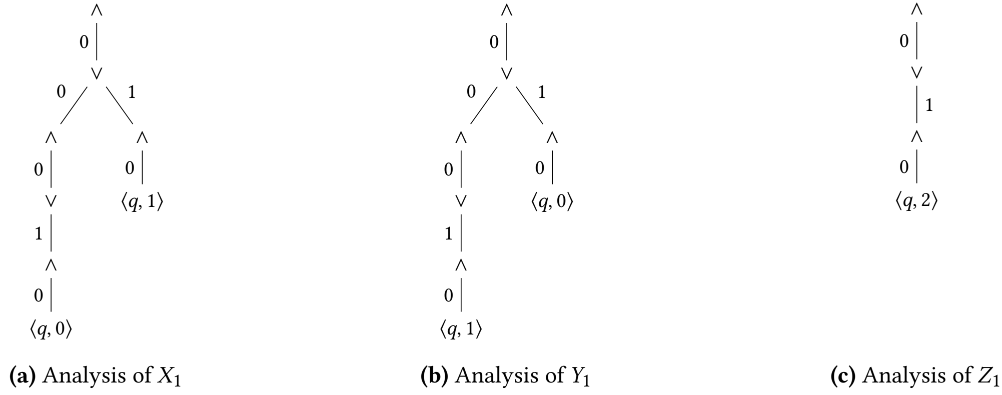
An intuition of the effect of this transformation in our setting, network verification, is that it unrolls the topology and replaces predicates on the global states of all the network elements by local predicates on the state of, for example, a given switch. Then, the state of the ports or the packets received by the other switches will not be considered to compute the output of the switch.

Example 7.1. Applying the partial instance and the predicate specialization to the program of Figure 2 transforms the definition of `linked(X,Y,P)` into a set of specialized predicates `linked_X_Y(P)` for each pair of linked locations `X` and `Y`. These new predicates are then described independently of the rest of the topology.

In other words, our transformations roughly rewrite generic, reusable and (usually) clear and short specifications into a network-specific form closer to NoD programs. Doing so by hand is obviously possible, but also lengthy, complex and error-prone, meaning that our certified optimizations conciliate performances and safety.

8 Related works

Many Datalog optimizations have been developed and formalized. The seminal [10] provides a general survey about Datalog, including some program transformation methods that annotate the program to help top-down, goal-oriented evaluations. More recent papers focus on evaluation technique [30] or user-directed and / or domain-specific optimizations [7, 32]. Zook et al. developed a static typing system for Datalog [37], but it only checks the type sanity of a program.



(a) Analysis of X_1 **(b)** Analysis of Y_1 **(c)** Analysis of Z_1

Figure 24. Analysis of a heterogeneously recursive program

Soufflé [31] is a static analysis tool using Datalog as a specification language. It performs Datalog-level optimizations, such as magic sets and user-directed rule inlinings. The Datalog code is translated into a relational algebra *via* the Futamura projection, where the interpreter is the semi-naive evaluation [2]. Although a form of specialization, this transformation – not discussed or illustrated – does not seem to produce an explicit analysis of the program value flows.

The idea of predicate specialization is taken from the code of [25] (the paper did not mention it). Our proposal contains, to our knowledge, the first explicit and formally defined value-focused static analysis and associated optimization for Datalog. Altough the idea was previously toyed with in [8, 31] as a debugging tool, our work is also, still to our knowledge, the first formalization of a Datalog trace semantics, not to mention a certified one.

The DataCert project aims at building a fully and deeply verified environment for data intensive management tools, the same way [21, 22] provide verified realistic C and ML compilers. Kriener et al. used Coq in [20] to prove the equivalence of different Prolog semantics. However, as far as we know, our work is the first formally proved implementation of non-trivial optimizations for a declarative and popular language, Datalog. It is also the first full-blown application of [5]. Although we had to slightly extend the formalization, our work shows that it can concretely be used to prove results on Datalog's use, giving credits to Datacert's ambition of offering a full environment for Datalog, among other aspects of data intensive applications.

9 Conclusion and perspectives

We design a static analysis and two transformations for Datalog, and prove their adequacy properties. To do so, we introduce a Datalog trace semantics with a certified operator. Citing [23], [5] comments on the relevance of MathComp to model Datalog, especially in the use of finTypes. Although the heavy use of dependent types to fit into the finType framework was especially cumbersome when manually adding and computing rules in existing programs, the ability to use paper-like set notations was most helpful. Compared to more traditional Coq definitions, they are clearer, easier to certify and to extract or translate into executable code.

Although this work was undertaken with the setting of network verification in mind, the optimizations are application-agnostic, and might be relevant for other areas. In general, we believe that our work was a good match for a first realistic experiment in formally proving Datalog optimizations, and that the tools and experience we developed now allow us or others to consider new, potentially more ambitious optimizations. In that spirit, we already laid out ideas for a more precise version of our static analysis.

On a more theoretical side, these ideas raise questions on a tighter characterization of recursion, which we also hope to address in the near future.

Acknowledgements

We would like to thank the anonymous reviewers for their numerous and helpful comments, which resulted in a dramatically enriched paper. We also thank Théo Winterhalter and Arthur Azevedo De Amorim for helping us getting started with some aspects of the Coq formalization.

References

[1] Coq development for this paper, https://orange-opensource.github.io/octant-proof

[2] Abiteboul, S., Hull, R., Vianu, V. (eds.): Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1995)

[3] 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: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1371–1382. ACM (2015)

[4] Bégay, P.L., Crégut, P., Monin, J.F.: Developing sequence and tree fintypes in MathComp. Coq Workshop 2020 (Jul 2020), https://coq-workshop.gitlab.io/2020/abstracts/Coq2020_03-03-seq-fintype.pdf

[5] Benzaken, V., Contejean, É., Dumbrava, S.: Certifying Standard and Stratified Datalog Inference Engines in SSReflect. In: International Conference on Interective Theorem Proving. Brasilia, Brazil (2017), https://hal.archives-ouvertes.fr/hal-01745566

[6] Benzaken, V., Contejean, É., Dumbrava, S.: Datalogcert. https://framagit.org/formaldata/datalogcert/ (2017)

[7] Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 243–262 (2009)

[8] Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A new proposal for debugging datalog programs. Electronic Notes in Theoretical Computer Science **216**, 79–92 (2008)

[9] Calì, A., Gottlob, G., Lukasiewicz, T.: Datalog±: a unified approach to ontologies and integrity constraints. In: Proceedings of the 12th International Conference on Database Theory. pp. 14–30. ACM (2009)

[10] Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE transactions on knowledge and data engineering **1**(1), 146–166 (1989)

[11] Chin, B., von Dincklage, D., Ercegovac, V., Hawkins, P., Miller, M.S., Och, F., Olston, C., Pereira, F.: Yedalog: Exploring knowledge at scale. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)

[12] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science **277**(1-2), 47–103 (2002)

[13] DeTreville, J.: Binder, a logic-based security language. In: Proceedings 2002 IEEE Symposium on Security and Privacy. pp. 105–113. IEEE (2002)

[14] Dumbrava, S.G.: Formalisation en Coq de Bases de Données Relationnelles et Déductives -et Mécanisation de Datalog. Ph.D. thesis, Université Paris-Sud (2016), http://www.theses.fr/2016SACLS525

[15] Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: The lixto data extraction project: back and forth between theory and practice. In: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 1–12. ACM (2004)

[16] Grumbach, S., Wang, F.: Netlog, a rule-based language for distributed programming. In: International Symposium on Practical Aspects of Declarative Languages. pp. 88–103. Springer (2010)

[17] Hoder, K., Bjørner, N., de Moura, L.: μz–an efficient engine for fixed points with constraints. In: International Conference on Computer Aided Verification. pp. 457–462. Springer (2011)

[18] Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. pp. 1213–1216. ACM (2011)

[19] Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: International Conference on Algebraic Methodology and Software Technology. pp. 258–273. Springer (2004)

[20] Kriener, J., King, A., Blazy, S.: Proofs you can believe in. proving equivalences between prolog semantics in coq. pp. 37–48 (09 2013). https://doi.org/10.1145/2505879.2505886

[21] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. In: ACM SIGPLAN Notices. vol. 49, pp. 179–191. ACM (2014)

[22] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009), http://xavierleroy.org/publi/compcert-CACM.pdf

[23] Libkin, L.: The finite model theory toolbox of a database theoretician. In: Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 65–76. ACM (2009)

[24] Lopes, N.P.: Network verification website (benchmarks and code). http://web.ist.utl.pt/nuno.lopes/netverif/

[25] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 499–512. USENIX Association, Oakland, CA (2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

[26] Lopes, N.P., Bjorner, N., Godefroid, P., Varghese, G.: Network verification in the light of program verification. Tech. rep., Microsoft (September 2013), https://www.microsoft.com/en-us/research/publication/network-verification-in-the-light-of-program-verification/

[27] Lu, L., Cleary, J.G.: An operational semantics of starlog. In: International Conference on Principles and Practice of Declarative Programming. pp. 294–310. Springer (1999)

[28] de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24

[29] Ramakrishnan, R., Ullman, J.D.: A survey of deductive database systems. The journal of logic programming **23**(2), 125–149 (1995)

[30] Ryzhyk, L., Budiu, M.: Differential datalog. In: Datalog (2019)

[31] Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 196–206. CC 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2892208.2892226, http://doi.acm.org/10.1145/2892208.2892226

[32] Shen, W., Doan, A., Naughton, J.F., Ramakrishnan, R.: Declarative information extraction using datalog with embedded extraction predicates. In: Proceedings of the 33rd international conference on Very large data bases. pp. 1033–1044. VLDB Endowment (2007)

[33] Sozeau, M.: Equations: A dependent pattern-matching compiler. In: International Conference on Interactive Theorem Proving. pp. 419–434. Springer (2010)

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

[35] Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 17–27 (2008)

[36] Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM (JACM) **23**(4), 733–742 (1976)

[37] Zook, D., Pasalic, E., Sarna-Starosta, B.: Typed datalog. In: International Symposium on Practical Aspects of Declarative Languages. pp. 168–182. Springer (2009)