Rule Rewriting Revisited: A Fresh Look at Static Filtering for Datalog and ASP

Philipp Hanisch
Knowledge-Based Systems Group, TU Dresden, Germany

Markus Krötzsch
Knowledge-Based Systems Group, TU Dresden, Germany

Abstract

Static filtering is a data-independent optimisation method for Datalog, which generalises algebraic query rewriting techniques from relational databases. In spite of its early discovery by Kifer and Lozinskii in 1986, the method has been overlooked in recent research and system development, and special cases are being rediscovered independently. We therefore recall the original approach, using updated terminology and more general filter predicates that capture features of modern systems, and we show how to extend its applicability to answer set programming (ASP). The outcome is strictly more general but also more complex than the classical approach: double exponential in general and single exponential even for predicates of bounded arity. As a solution, we propose tractable approximations of the algorithm that can still yield much improved logic programs in typical cases, e.g., it can improve the performance of rule systems over real-world data in the order of magnitude.

2012 ACM Subject Classification Theory of computation → Logic and databases; Theory of computation → Constraint and logic programming; Theory of computation → Equational logic and rewriting

Keywords and phrases Rule rewriting, static optimisation, static filtering, Datalog, Answer Set Programming

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.5

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2601.05108` [16]

Funding This work is partly supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project number 389792660 (TRR 248, Center for Perspicuous Systems) and 390696704 (CeTI Cluster of Excellence) and by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR, Federal Ministry of Research, Technology and Space) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) and in DAAD project 57616814 (SECAI, School of Embedded Composite AI) as part of the program Konrad Zuse Schools of Excellence in Artificial Intelligence.

1 Introduction

Besides its many other advantages, the declarative nature of logic-based rule languages also enables effective optimisation through logically equivalent rewritings. Of course, already for plain Datalog, logical equivalence is undecidable, and highly complex in decidable special cases [6]. But many concrete transformations that guarantee logical equivalence have been proposed, ranging from the popular *magic sets* [3, 28, 29] to recent proposals [31, 32, 35, 36]. Like these examples, many rule rewritings are *static* optimisations, which do not depend on the concrete set of facts (data) that is to be processed.

One of the classical proposals of this kind is *static filtering*, the generalisation of *selection pushing* methods from relational databases to Datalog, introduced by Kifer and Lozinskii [22, 23, 24]. The underlying principle of enforcing restrictions ("filters") on intermediate results as early as possible in the computation is a tried and tested paradigm in databases,

Table 1 Runtimes for example program (1)–(3) with $\ell = 19$ (median of five runs, timeout at 5min, evaluation system: Linux, AMD Ryzen 7 PRO 5850U, 16 GiB RAM).

	Soufflé v2.5	**Nemo** v0.8.1	**Clingo** v5.8.0	**DLV** v2.1.2
Original	1214ms	>5min	1104ms	74579ms
Rewritten	24ms	99ms	8ms	3ms

and does – in contrast to, e.g., magic sets – not change the general structure of derivations fundamentally. Moreover, static filtering strongly increases the effectivity of the related method of projection pushing [24]. Static filtering can enable polynomial (data complexity) or exponential (combined complexity) performance improvements (see Examples 1 and 23).

Surprisingly, the journal paper of Kifer and Lozinskii has attracted less than 50 citations in the past 35 years.[1] Actual uses of the method are rarely reported [5, 11, 26], while most mentions consider it distantly related work. Is this basic optimisation approach maybe so fundamental that it is not even mentioned by implementers? Or is it known and used under another name?

▶ **Example 1.** We ran a small experiment to find out, using the following Datalog rules (see Section 2 for a formal introduction to Datalog, and the literature for more details [25, 27]):

$$p(0,\dots,0,0,\mathtt{a}). \qquad p(1,\dots,1,0,\mathtt{b}). \tag{1}$$

$$p(x_1,\dots,x_i,1,0,\dots,0,y) \leftarrow p(x_1,\dots,x_i,0,1,\dots,1,y) \qquad \text{for all } i \in \{1,\dots,\ell\} \tag{2}$$

$$\mathit{out}(y) \leftarrow p(x_1,\dots,x_\ell,y) \wedge y \doteq \mathtt{b} \tag{3}$$

where p is an $(\ell+1)$-ary predicate with $\ell \geq 1$, $\doteq$ denotes equality, y and x_k are variables, and 0, 1, `a`, and `b` are constants. Rules (2) implement a binary counter over ℓ bits, so exponentially many p-facts are inferred from the facts (1). Optimisation is possible if we are only interested in inferences for predicate *out*: then the precondition $y \doteq \mathtt{b}$ can be added to the rules (2), so that just one new p-fact follows. Static filtering produces this rewriting.

Many modern rule systems let users specify output predicates. For our experiment, we considered Datalog engines Soufflé [20] (syntax `.output out`) and Nemo [18] (syntax `@export out :- csv{}.`), and ASP engines Clingo [13] and DLV [2] (syntax `#show out/1.` for both). Table 1 shows runtimes for the original program and the optimised version. Evidently, each tool benefits from the optimisation, yet none implements it by default.

Why is such a natural optimisation, considered standard in relational query optimisation, ignored in modern rule systems? Research culture may be a reason. Typical benchmarks for comparing systems are already optimised, so static optimisations offer no benefits. They are likely more effective with less polished user inputs, especially during development and experimentation. Moreover, static filtering has not attained the popularity of other approaches, especially magic sets and semi-naive evaluation, and may not be known to many implementers. The original description relies on *system graphs* as an auxiliary concept that many readers may not know today, yet we are not aware of modern accounts or textbook explanations of the method.

[1] Google Scholar, `https://scholar.google.com/scholar?cites=13499523163799224695`, retrieved 8 September 2025

Another reason might be practicality. Kifer and Lozinskii found the method to be exponential in the worst case – as hard as the Datalog reasoning task it aims to optimise – and did not suggest tractable variants. It is also left open how static filtering generalises to further filter expressions (Kifer and Lozinskii only consider binary relations like $=$, $\neq$, and $\leq$), and to non-monotonic negation or ASP.

In this work, we therefore revisit static filtering and introduce a generalised rewriting that supports arbitrary filters (Section 3). Analysing the complexity of our method (Section 4), we find further exponential increases over the prior special case, which motivates our design of a tractable variant that is still reasonably general (Section 5). Finally, we show how static filtering can be extended to rules with negation and to ASP (Section 6), before comparing closely related works (Section 7). Detailed proofs for all claims are found online [16].

2 Preliminaries

We consider a signature based on mutually disjoint, countably infinite sets of *constants* $\mathbf{C}$, *variables* $\mathbf{V}$, and *predicates* $\mathbf{P}$, where each predicate $p \in \mathbf{P}$ has *arity* $\mathsf{ar}(p) \geq 0$. *Terms* are elements of $\mathbf{V} \cup \mathbf{C}$. We write bold symbols $\boldsymbol{t}$ for lists $t_1, \ldots, t_{|\boldsymbol{t}|}$ of terms (and for terms of special types, such as lists of variables $\boldsymbol{x}$). Lists are treated like sets when order is not relevant, so we may write, e.g., $\boldsymbol{x} \subseteq \mathbf{V}$. By $\mathsf{var}(E)$ we generally denote the set of variables in an expression E. For a mapping $\sigma\colon \boldsymbol{x} \to \mathbf{C}$ and expression E, we obtain $E\sigma$ by simultaneously replacing all occurrences of $x \in \boldsymbol{x}$ by $\sigma(x)$.

Rules and programs

An *atom* $\underline{a}$ is an expression $p(\boldsymbol{t})$ with $p \in \mathbf{P}$, $\boldsymbol{t}$ a list of terms, and $\mathsf{ar}(p) = |\boldsymbol{t}|$. A (Datalog) *rule* ρ is a formula $H \leftarrow B$, where the *head* H is an atom, the *body* B is a conjunction of atoms, and all variables are implicitly universally quantified. We require that all variables in H also occur in B (*safety*). Conjunctions of atoms may be treated as sets of atoms. A (Datalog) *program* P is a finite set of rules. A predicate p is an *EDB atom* in P if it only occurs in rule bodies, and an *IDB atom* if it occurs in some rule head.[2]

Semantics

A *fact* for predicate p is an atom $p(\boldsymbol{c})$ with $\boldsymbol{c} \subseteq \mathbf{C}$. A *database* $\mathcal{D}$ for a program P is a potentially infinite set of facts for EDB predicates of P. We allow $\mathcal{D}$ to be infinite, so as to accommodate conceptually infinite built-in relations, such as $\leq$. Practical systems typically evaluate such built-ins on demand and syntactically ensure that infinite built-ins do not lead to infinite derivations, e.g., by requiring that variables in built-ins also occur in body atoms with finite predicates. Such concerns are unimportant to our results, so we can unify (given) input facts and (computed) built-in relations.

The (unique) *model* $\mathcal{M}$ for program P and database $\mathcal{D}$ is the least set of facts such that (1) $\mathcal{D} \subseteq \mathcal{M}$, and (2) for every rule $\rho \in P$ with variables $\boldsymbol{x}$, and every mapping $\sigma : \boldsymbol{x} \to \mathbf{C}$, if $\sigma(B) \subseteq \mathcal{M}$ then $\sigma(H) \subseteq \mathcal{M}$. If $\sigma(B) \subseteq \mathcal{M}$, we also call σ a *match* of ρ on $\mathcal{M}$. Models can be equivalently defined through iterated rule applications, proof trees, or as least models of P viewed as a first-order logic theory [1].

For a rule $H \leftarrow B$ with variables $\boldsymbol{x}$, a mapping $\sigma : \boldsymbol{x} \to \mathbf{C}$ is *applicable* to a database $\mathcal{D}$ if $B\sigma \subseteq \mathcal{D}$ and $H\sigma \nsubseteq \mathcal{D}$. A set of facts $\mathcal{D}$ is *closed under a program* P, written $\mathcal{D} \models P$, if there is no $\rho \in P$ with an applicable mapping σ. For a predicate p, let $p^{\mathcal{D}} = \{\boldsymbol{c} \mid p(\boldsymbol{c}) \in \mathcal{D}\}$ denote the tuples of p-facts in $\mathcal{D}$.

[2] These terms originate from *extensional/intentional database*.

Normal form

To simplify presentation, we require that rules contain only variables, no constants, and that atoms in rules do not contain repeated variables. This normal form can be established by defining auxiliary EDB predicates $(\_ \doteq \_)^{\mathcal{D}} = \{\langle \mathtt{c}, \mathtt{c}\rangle \mid \mathtt{c} \in \mathbf{C}\}$ and $(\_ \doteq \mathtt{d})^{\mathcal{D}} = \{\mathtt{d}\}$ for every constant $\mathtt{d} \in \mathbf{C}$. Now every occurrence of $\mathtt{d} \in \mathbf{C}$ in a rule is replaced by a fresh variable x, and the atom $x \doteq \mathtt{d}$ is added to the rule body. Similarly, every non-first occurrence of a variable x in a body atom is replaced by a fresh variable x', and the atom $x \doteq x'$ is added to the rule body. Similar normalisations can be used for any built-in function that reasoners might support, e.g., arithmetic functions as in a rule $p(x + y) \leftarrow q(x, y)$ can be rewritten as $p(z) \leftarrow q(x, y) \wedge z \doteq x{+}y$ with $(\_ \doteq \_{+}\_)^{\mathcal{D}} = \{\langle \mathtt{l}, \mathtt{m}, \mathtt{n}\rangle \in \mathbf{C}^3 \mid \mathtt{l} = \mathtt{m} + \mathtt{n}\}$. As before, the infinite EDB predicate is just a conceptual model for a real systems' on-demand computation of its supported built-in functions.

Outputs and filters

Given a program P, we consider distinguished sets $\mathbf{P}_{\mathsf{out}} \subseteq \mathbf{P}$ of *output predicates* and $\mathbf{F} \subseteq \mathbf{P}$ of *filter predicates*, where all filter predicates must be EDB predicates in P. Outputs define which inferences are of interest to users, and are supported in many reasoners, including the ones in Table 1. Filters are used in our algorithms to reduce inferences for non-output predicates, and would be defined by the reasoner implementation: they should be easy to check and highly selective. For example, the above predicates for $\doteq$ are good filters, whereas an EDB predicate that is stored in a large file on disk most likely is not. However, specific predicates like "x is a string with letter e in third position" or pre-loaded data with fast index structures can also be suitable filters. Given a set of atoms B, we define $B_{\mathbf{F}} = \{p(\boldsymbol{t}) \in B \mid p \in \mathbf{F}\}$ and $B_{\bar{\mathbf{F}}} = B \setminus B_{\mathbf{F}}$ as the conjunction of atoms with filter predicates and, respectively, atoms with non-filter predicates.

Rules with generalised filters

A *rule with generalised filter expressions* has the form $H \leftarrow B_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}}$ with H a head atom, $B_{\bar{\mathbf{F}}}$ a conjunction of non-filter atoms, and $G_{\mathbf{F}} \in \mathbf{G}$ a positive boolean combination of filter atoms, defined recursively:

$$\mathbf{G} ::= p(\boldsymbol{t}) : p \in \mathbf{F}, |\boldsymbol{t}| = \mathsf{ar}(p), \boldsymbol{t} \subseteq \mathbf{C} \cup \mathbf{V} \mid (\mathbf{G} \wedge \mathbf{G}) \mid (\mathbf{G} \vee \mathbf{G}).$$

Positive boolean combinations of body atoms are normally syntactic sugar in Datalog: we can replace any (body) disjunction $A[\boldsymbol{x}] \vee B[\boldsymbol{y}]$ over (possibly overlapping) sets of variables $\boldsymbol{x}$ and $\boldsymbol{y}$ by a fresh atom $D[\boldsymbol{x} \cup \boldsymbol{y}]$, and add rules $D[\boldsymbol{x} \cup \boldsymbol{y}] \leftarrow A[\boldsymbol{x}]$ and $D[\boldsymbol{x} \cup \boldsymbol{y}] \leftarrow B[\boldsymbol{y}]$. The fresh atom D avoids the exponential blow-up that would occur if we would instead create two copies of the rule, one with A and one with B. With potentially infinite filter predicates, however, this syntactic transformation is not natural, since the auxiliary D could be infinite, whereas it is easy for systems to evaluate nested expressions $G_{\mathbf{F}}$ in place. Therefore, if not otherwise stated, all *programs* below may include generalised filters. Their normal form is defined as for Datalog.

3 Static Filtering: A General Algorithm

Next, we present a method for optimising Datalog programs by static rewriting. The optimised programs have smaller least models that are nonetheless guaranteed to contain the same facts for output predicates $\mathbf{P}_{\mathsf{out}}$. A comparison with the work of Kifer and Lozinskii is given in Section 7.

We consider a fixed program P in normal form, a database $\mathcal{D}$, filter predicates $\mathbf{F}$, and output predicates $\mathbf{P}_{\mathsf{out}}$. Facts for non-filter predicates in $\mathcal{D}$ are irrelevant for static filtering.

▶ **Example 2.** As a running example, we consider a depth-bounded reachability check:

$$r(x, y, n) \leftarrow e(x, y) \wedge n \doteq 0 \tag{4}$$

$$r(x, z, m) \leftarrow r(x, y, n) \wedge e(y, z) \wedge m \doteq n{+}1 \tag{5}$$

$$\mathit{out}(y) \leftarrow r(x, y, n) \wedge x \doteq \mathsf{a} \wedge n \leq 5 \tag{6}$$

where *out* is the output predicate. Notably, rule (5) can produce infinitely many inferences if the graph described by e is cyclic, but only finitely many nodes reachable from a in ≤ 5 steps are relevant for the output.

The logic of filters

For a given arity $k > 0$, let $\mathbf{N}_k = \{\boxed{\mathbf{1}}, \dots, \boxed{\boldsymbol{k}}\}$ be a set of k positional markers. A *filter atom* (for arity k) is an expression $\underline{f} = p(\boxed{\boldsymbol{m_1}}, \dots, \boxed{\boldsymbol{m_\ell}})$ where $p \in \mathbf{F}$ with $\ell = \mathsf{ar}(p)$ and $\boxed{\boldsymbol{m_i}} \in \mathbf{N}_k$ for $i = 1, \dots, \ell$. The semantics of $\underline{f}$ is the relation $\underline{f}^{\mathcal{D}} = \{\boldsymbol{c} \in \mathbf{C}^k \mid \langle c_{m_1}, \dots, c_{m_\ell} \rangle \in p^{\mathcal{D}}\}$. Let $\mathbf{F}[k]$ be the set of all filter atoms of arity k over $\mathbf{F}$. The *filter formulas* $\mathcal{F}_k$ are the positive boolean formulas over $\mathbf{F}[k]$:

$$\mathcal{F}_k ::= \mathbf{F}[k] \mid \top \mid \bot \mid (\mathcal{F}_k \wedge \mathcal{F}_k) \mid (\mathcal{F}_k \vee \mathcal{F}_k) \tag{7}$$

Their semantics is defined as expected: $\top^{\mathcal{D}} = \mathbf{C}^k$, $\bot^{\mathcal{D}} = \emptyset$, $(F \wedge G)^{\mathcal{D}} = F^{\mathcal{D}} \cap G^{\mathcal{D}}$, and $(F \vee G)^{\mathcal{D}} = F^{\mathcal{D}} \cup G^{\mathcal{D}}$. Given filter formulas $F, G \in \mathcal{F}_k$, we write $F \models G$ if $F^{\mathcal{D}} \subseteq G^{\mathcal{D}}$, and $F \equiv G$ if $F \models G$ and $G \models F$. We can assume that $\bot$ and $\top$ are only used at the root level, using the usual simplifications: $(\bot \wedge F) \mapsto \bot$, $(\top \wedge F) \mapsto F$, $(\bot \vee F) \mapsto F$, and $(\top \vee F) \mapsto \top$ (and their commutated versions). A filter formula is *simplified* if none of these rewritings applies to it.

Example 2 might use filter predicates $\_ \doteq \mathsf{a}$, $\_ \leq 5$, and $\_ \doteq \_{+}1$. Since filter formulas do not allow constants, we assume distinct predicates for every pattern of constant use. Implementations generally decide which filters to consider – those that occur syntactically are required, but others can be added.

Optimised filter computation

For every IDB predicate $p \in \mathbf{P}$, we seek a filter formula $\mathsf{flt}(p) \in \mathcal{F}_{\mathsf{ar}(p)}$, such that we only need to derive facts $p(\boldsymbol{c})$ if $\boldsymbol{c} \in \mathsf{flt}(p)^{\mathcal{D}}$. To obtain an algorithm, some operations have to be concretely implemented for the chosen filters $\mathbf{F}$ and every $k \geq 1$:

1. It must be possible to decide $F \models G$ for any $F, G \in \mathcal{F}_k$.
2. There is a canonical representation function $\mathsf{rep} : \mathcal{F}_k \to \mathcal{F}_k$, such that $\mathsf{rep}(F) \equiv F$, and $F \equiv G$ implies $\mathsf{rep}(F) = \mathsf{rep}(G)$, for all $F, G \in \mathcal{F}_k$.

For an atom $p(\boldsymbol{x})$ with arity $\mathsf{ar}(p) = k$, the mapping $\iota_{p(\boldsymbol{x})} : \boxed{\boldsymbol{i}} \mapsto x_i$ maps the positional markers $\boxed{\mathbf{1}}, \dots, \boxed{\boldsymbol{k}}$ to $\boldsymbol{x}$. We extend $\iota_{p(\boldsymbol{x})}$ to filter formulas F, i.e., we obtain $\iota_{p(\boldsymbol{x})}(F)$ by replacing each positional marker $\boxed{\boldsymbol{i}}$ with $\iota_{p(\boldsymbol{x})}(\boxed{\boldsymbol{i}}) = x_i$. For $r(x, y, n)$ of rule (5) and $F = \boxed{\mathbf{3}} \leq 5$, e.g., we get $\iota_{r(x,y,n)}(F) = \iota_{r(x,y,n)}(\boxed{\mathbf{3}} \leq 5) = n \leq 5$.

Algorithm 1 describes the iterative computation of optimised filters. Line L2 initialises filters to $\top$ for output predicates ("compute all facts"), and to $\bot$ for all others. The formulas are then generalised by looping over all rules (L4) and their non-filter body atoms (L5). Matches to a rule can be restricted to those that satisfy both the head predicate's filter $\mathsf{flt}(h)$

Algorithm 1 Static filter computation.

Input: program P, output predicates $\mathbf{P}_{\mathsf{out}}$
Output: filter formulas $\mathsf{flt}(p)$ for IDB predicates p

```
1 for p ∈ P where p is an IDB predicate do
2     if p ∈ P_out then flt(p) := rep(⊤) else flt(p) := rep(⊥)
3 repeat
4     for ρ ∈ P with ρ = h(x) ← B_F̄ ∧ G_F do
5         for b(y) ∈ B_F̄ where b is an IDB predicate do
6             G := ι_{h(x)}(flt(h)) ∧ G_F
7             M := ⋀{F ∈ F_{ar(b)} | G ⊨ ι_{b(y)}(F)}
8             flt(b) := rep(flt(b) ∨ M)
9 until all formulas flt(p) remain unchanged
```

and the rule's own filter expression $G_{\mathbf{F}}$, which are combined into a filter formula $G \in \mathcal{F}_{|\mathsf{var}(\rho)|}$ (L6); we map $\mathsf{flt}(h)$ to the variables used in the rule. To find the strongest filter formula M for body atom $b(\boldsymbol{y})$, we take a conjunction of all filters $F \in \mathcal{F}_{\mathsf{ar}(b)}$ that follow from G when mapping F to variables $\boldsymbol{y}$ as used in $b(\boldsymbol{y})$ (L7). Finally, we generalise the current filter for b by including the new M disjunctively, and taking the canonical representation (L8).

▶ **Example 3.** We apply Algorithm 1 to Example 2, with a semantically minimised canonical representation in disjunctive normal form. Initially, we have $\mathsf{flt}(\mathit{out}) = \top$ and $\mathsf{flt}(r) = \bot$. Processing body atom $r(x, y, n)$ of rule (6), we get $G = \top \wedge x \doteq \mathsf{a} \wedge n \leq 5$. Therefore, we have $G \models x \doteq \mathsf{a} = \iota_{r(x,y,n)}(\boxed{1} \doteq \mathsf{a})$ and $G \models n \leq 5 = \iota_{r(x,y,n)}(\boxed{3} \leq 5)$, and M is equivalent to the conjunction $\boxed{1} \doteq \mathsf{a} \wedge \boxed{3} \leq 5$. This is also the new filter condition $\mathsf{flt}(r)$.

In the next iteration, for body atom $r(x, y, n)$ in rule (5), we get $G = x \doteq \mathsf{a} \wedge m \leq 5 \wedge m \doteq n{+}1$. The only relevant entailments are $G \models x \doteq \mathsf{a} = \iota_{r(x,y,n)}(\boxed{1} \doteq \mathsf{a})$ and $G \models n \leq 5 = \iota_{r(x,y,n)}(\boxed{3} \leq 5)$, so we obtain the same M and $\mathsf{flt}(r)$ as before. The algorithm then terminates.

Correctness and use in optimisation

Algorithm 1 is correct in the sense that rule applications can be safely restricted to applying rules only when the conclusion $p(\boldsymbol{c})$ satisfies the computed filter formula $\mathsf{flt}(p)$, as there is no risk of changing derivations for outputs. However, adding body atoms for $\mathsf{flt}(p)$ to all rules is often redundant. The next definition characterises choices for equivalent filter conditions.

▶ **Definition 4.** *Consider a rule $\rho = h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}} \in P$. Let $F_+ = \iota_{h(\boldsymbol{x})}(\mathsf{flt}(h)) \wedge G_{\mathbf{F}}$ denote the formula that combines the given filters $G_{\mathbf{F}}$ with the computed filter for $h(\boldsymbol{x})$, let $\mathbf{P}_{\mathsf{IDB}}$ denote the IDB predicates of P, and let $F_- = \bigwedge\{\iota_{q(\boldsymbol{y})}(\mathsf{flt}(q)) \mid q(\boldsymbol{y}) \in B_{\bar{\mathbf{F}}}, q \in \mathbf{P}_{\mathsf{IDB}}\}$ denote the filter formula that combines the computed filters for IDB atoms in $B_{\bar{\mathbf{F}}}$. Then a formula ψ is* admissible for ρ *if*

$$F_+ \models \psi \qquad \text{and} \qquad \psi \wedge F_- \models F_+ \,.$$

Then the rule $h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge \psi$ is an admissible rewriting *of ρ.*[3] *An* admissible rewriting *of P is a set that contains an admissible rewriting of each rule of P.*

[3] Technically, we simplify ψ: if $\psi = \bot$, the rewritten rule is deleted instead; if $\psi = \top$, then ψ is omitted.

Algorithm 2 Computing admissible filters.

Input: rule ρ, formulas F_+ and F_- as in Definition 4
Output: formula ψ admissible for ρ
10 $\psi \coloneqq F_+$
11 **for** *every occurrence o of an atom in ψ* **do**
12 **if** $F_- \wedge \psi[o \mapsto \top] \models \psi$ **then** $\psi \coloneqq \psi[o \mapsto \top]$
13 **return** simplification of ψ

The conditions of Definition 4 ensure that admissible filters are equivalent to the canonically extended body filter F_+ under the assumption that body atoms are also restricted to their computed filters F_-. We obtain the following.

▶ **Theorem 5.** *If P' is an admissible rewriting of P, and $p(\boldsymbol{c})$ is a fact with $p \in \mathbf{P}_{\mathsf{out}}$, then $P, \mathcal{D} \models p(\boldsymbol{c})$ iff $P', \mathcal{D} \models p(\boldsymbol{c})$.*

▶ **Example 6.** Using the filters computed in Example 3, we can get this admissible rewriting:

$$r(x, y, n) \leftarrow e(x, y) \wedge n \doteq 0 \wedge x \doteq \mathsf{a} \tag{8}$$

$$r(x, z, m) \leftarrow r(x, y, n) \wedge e(y, z) \wedge m \doteq n{+}1 \wedge m \leq 5 \tag{9}$$

$$\mathit{out}(y) \leftarrow r(x, y, n) \tag{10}$$

For rule (8), $n \doteq 0 \wedge x \doteq \mathsf{a}$ is admissible, as $n \doteq 0 \models n \leq 5$. For rule (9), body atom $r(x, y, n)$ yields $F_- = x \doteq \mathsf{a} \wedge n \leq 5$, while $F_+ = x \doteq \mathsf{a} \wedge m \leq 5 \wedge m \doteq n + 1$; so $m \doteq n{+}1 \wedge m \leq 5$ is indeed admissible. Finally, for rule (10), an empty ($\top$) filter is admissible, since F_- already entails all necessary conditions. Importantly, the final rewriting (and any other admissible rewriting) will always terminate, even when Example 2 fails to do so.

Note that F_+ as given in Definition 4 is always admissible, but there can be much simpler expressions. Algorithm 2 shows a practical way to find good admissible filters, which would also find the rewriting of Example 6. An *occurrence* o of an atom in ψ refers to a single position in ψ where some atom is found (even if the same atom occurs more than once), and $\psi[o \mapsto \top]$ denotes the result of replacing this occurrence by $\top$. Each iteration in line L12 preserves admissibility of ψ, and the algorithm terminates after linearly many iterations.

Cost and benefits of static filtering

While the concrete performance impact of the rewriting depends on P and $\mathcal{D}$, static filtering is guaranteed to only reduce but never increase the number of logical entailments. The next result is a direct consequence of the proof of Theorem 5, where we showed that $\mathcal{M}' = \{q(\boldsymbol{d}) \in \mathcal{M} \mid \boldsymbol{d} \in \mathsf{flt}(q)^{\mathcal{D}}\} \cup \mathcal{D}$:

▶ **Theorem 7.** *Let P' be an admissible rewriting of P, and let $\mathcal{M}$ (resp. $\mathcal{M}'$) be the model of $\mathcal{D}$ and P (resp. of $\mathcal{D}$ and P'). Then $\mathcal{M}' \subseteq \mathcal{M}$, and every match of a rule $\rho' \in P'$ on $\mathcal{M}'$ is also a match of the corresponding rule $\rho \in P$ on $\mathcal{M}$.*

In particular, every derivation (proof tree) over $\mathcal{D}$ and P' corresponds to a analogous derivation over $\mathcal{D}$ and P, and the bottom-up computation of $\mathcal{M}'$ requires at most as many rule applications as the bottom-up computation of $\mathcal{M}$. The additional cost associated with the use of P' is limited to the cost of checking, for each rule match, if the rewritten filters

Table 2 Worst-case number of computing steps of Algorithm 1 in several scenarios, with details on upper and lower bounds depending on the filter size and head arity.

Filter arity	Filter size	Head arity		#Steps	
variable	infinite	variable	$\leq$	doubly exponential	Thm 11
	doubly exponential	constant ≥ 2	$\geq$		Prop 12
constant ≥ 2	infinite	variable	$\leq$	exponential	Thm 11
	polynomial		$\geq$		Ex 9
variable	infinite	constant $= 1$	$\leq$	exponential	Thm 11
constant ≥ 2	exponential		$\geq$		Prop 13
variable	exponential	variable	$\leq$	exponential	Thm 14
constant ≥ 2	polynomial	variable	$\geq$		Ex 9
variable	polynomial	constant ≥ 2	$\leq$	polynomial	Thm 14

rather than the original filters are satisfied. This cost can be controlled by system designers through the choice of filter predicates to push. The potential savings, on the other hand, can be in the order of $|\mathcal{M}|$.

The fact that static filtering preserves the structure of rules, programs, and derivations also improves understandability by human users and compatibility with other optimisations, be it logical (e.g., magic sets) or operational (e.g., join-order optimisation). The rewriting is even *idempotent*, i.e., already optimised programs will not be modified further when applying static filtering again. None of these advantages is common to all logic program optimisation methods, a prominent counterexample being magic sets.

4 Lower and Upper Bounds for Termination

Next, we analyse the time complexity of Algorithm 1 in terms of the number of iterations. For now, we abstract from the complexity of checking $\models$ and computing representatives rep, which are discussed in more detail in Section 5. Our results are summarised in Table 2, depending on given bounds on predicate arity and size of filter relations, where *variable* arity and *infinite* size are least restrictive.

▶ **Example 8.** Kifer and Lozinskii [24] find that static filtering is exponential, and they give the following example:

$$r(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y) \tag{11}$$

$$r(\boldsymbol{x}_{i \rightleftharpoons j}, y) \leftarrow r(\boldsymbol{x}, y) \qquad \text{for all } 1 \leq i < j \leq |\boldsymbol{x}| \tag{12}$$

$$out(y) \leftarrow r(\boldsymbol{x}, y) \wedge \textstyle\bigwedge_{i=1}^{|\boldsymbol{x}|} x_i \doteq \mathsf{a}_i \tag{13}$$

where $\boldsymbol{x}_{i \rightleftharpoons j}$ denotes $\boldsymbol{x}$ with variables x_i and x_j swapped, and all a_i are constants. Rules (12) derive all (exponentially many) permutations of the tuples in r, and the filter computed for r must allow all permutations of $\boldsymbol{a}$. Rewriting to plain Datalog, we get rules $r(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y) \wedge \bigwedge_{i=1}^{|\boldsymbol{x}|} x_i \doteq \mathsf{a}_{\nu(i)}$ for exponentially many permutation functions ν.

Example 8 seems to establish an exponential lower bound for Algorithm 1, but the exponential complexity in this case stems merely from the representation of filter formulas. All permutations are obtained in linearly many pairwise swaps, and Algorithm 1 therefore terminates after linearly many iterations of loop L3. Restricting to the filter predicates in the

example, the result does require an exponential filter formula, but there are also filter logics that support polynomial representations, e.g., by simply introducing filter predicates for statements like "$\boldsymbol{b}$ is a permutation of $\boldsymbol{a}$". Example 8 therefore is not illustrating exponential behaviour in general, but we can find other examples that do:

▶ **Example 9.** Let p be a non-filter predicate of arity $\ell + 1$. For readability, we do not normalise the rules:

$$p(\boldsymbol{x}, y) \leftarrow e(\boldsymbol{x}, y) \tag{14}$$

$$p(x_1, \dots, x_i, 1, 0, \dots, 0, y) \leftarrow p(x_1, \dots, x_i, 0, 1, \dots, 1, y) \qquad \text{for all } i \in \{1, \dots, \ell\} \tag{15}$$

$$\mathit{out}(y) \leftarrow p(1, \dots, 1, y) \tag{16}$$

Rule (14) can be rewritten to require each variable in $\boldsymbol{x}$ to map to a constant from $\{0, 1\}$, which can be expressed in a compact generalised filter formula (with nested $\vee$). However, the exponentially many admissible combinations of lists from $\{0, 1\}^\ell$ are computed in Algorithm 1 by iterating over the successor relation for binary numbers, realised by rules (15). This requires exponentially many steps. Note that normalisation and static filtering in this case would only use filter predicates $\sqcup \doteq \sqcup$, $\sqcup \doteq 0$, and $\sqcup \doteq 1$, as defined in Section 2.

In fact, as we will see below, Algorithm 1 may even require a doubly exponential number of iterations, and still exponentially many for predicates of bounded arity. The key to showing corresponding upper bounds is the following lemma.

▶ **Lemma 10.** *Given n_h distinct head predicates, if Algorithm 1 performs $\geq n_h \cdot s$ iterations of loop L3, then there is a head predicate p, and a chain $F_1 \models \cdots \models F_s$ of non-equivalent filter formulas $F_i \in \mathcal{F}_{\mathsf{ar}(p)}$.*

In other words, (doubly) exponentially long runs require (doubly) exponentially "deep" filter logics. The number of available filter formulas yields a first major upper bound.

▶ **Theorem 11.** *Let there be $n_{\mathbf{F}}$ filter predicates of arity $\leq a_{\mathbf{F}}$, and n_h head predicates of arity $\leq a_h$. Then Algorithm 1 terminates after at most $n_h \cdot 2^{n_{\mathbf{F}} \cdot {a_h}^{a_{\mathbf{F}}}}$ many iterations.*

The doubly exponential worst case of Theorem 11 can be reached. As the theorem suggests, the arity of head predicates can even be constant, as long as it is > 1.

▶ **Proposition 12.** *There are programs with filter arity $a_{\mathbf{F}} = \ell$ for which Algorithm 1 requires 2^{2^ℓ} many iterations, even if the head arity is fixed and the size of the filter relations is at most doubly exponential.*

Theorem 11 shows that the upper bound drops to single exponential if either (1) we impose a constant bound on the arity $a_{\mathbf{F}}$ of filter predicates, or (2) we fix the arity a_h of the head predicates to $a_h = 1$. Example 9 showed exponential behaviour in case (1). We can also reach this upper bound in case (2).

▶ **Proposition 13.** *There are programs with head arity $a_h = 1$ and filter arity $a_{\mathbf{F}} = \ell$ for which Algorithm 1 requires 2^ℓ many iterations, even if the size of the filter relations is at most exponential.*

Proposition 12 and 13 consider filter relations that are of a size that is proportional to the double and single exponential length of the runs, whereas Example 9 only requires polynomially sized filter relations. The following result clarifies how filter relation cardinality may affect upper bounds.

▶ **Theorem 14.** *Let there be $n_{\mathbf{F}}$ filter predicates that correspond to relations in $\mathcal{D}$ of cardinality $\leq c_{\mathbf{F}}$, and let there be n_h head predicates of arity $\leq a_h$. Then Algorithm 1 terminates after at most $n_h \cdot ((n_{\mathbf{F}} \cdot c_{\mathbf{F}})^{a_h} + 2)$ iterations.*

Theorem 14 yields a polynomial upper bound for the case that filter relations are polynomially bounded and non-filter predicates have a fixed arity. However, many filters in existing systems correspond to infinite relations, so further approaches are required to make static filtering tractable in practice.

5 Tractable Static Filtering

In this section, we further analyse the complexity of Algorithm 1, and propose simplifications for making it tractable. Section 4 showed that runtimes can be prohibitive, even without considering the cost of individual operations. For a full analysis, we must also analyse the cost of lines L7 and L8. Since the result of L8 remains the same when replacing M by any $M' \equiv M$, implementations can optimise L7 by computing a potentially smaller M'. Nevertheless, Example 8 shows a case where every such M' still grows exponentially during polynomially many iterations. Our proposed solution is to restrict $\mathsf{flt}(p)$ to special forms of filter formulas that merely approximate those in Algorithm 1.

First, however, we need to address the problem that even the entailment of individual filter formulas is generally undecidable, even for common filters.

▶ **Proposition 15.** *If $\mathbf{F}$ contains predicates that can express arithmetic equalities $x = y + z$ and $x = y \cdot z$ over natural numbers, and $x = n$ for all $n \in \mathbb{N}$, then there is no algorithm that decides $F \models G$ for arbitrary $k > 0$ and $F, G \in \mathcal{F}_k$.*

It is well-known that arithmetic predicates are challenging to reason with, and Datalog engines commonly restrict to *safe* arithmetics, where all numeric variables are bound to finite extensions of other predicates. However, this restriction does not simplify static filtering, where abstract filter formulas are considered without such concrete bindings.

We therefore consider entailment relations that are sound but not necessarily complete, except for the basic semantics of the propositional operators in filter formulas.

▶ **Definition 16.** *Every filter formula F can be considered as a propositional logic formula over its filter atoms. Let $\models_{prop}$ denote the usual propositional logic entailment relation over these formulas. A binary relation $\mathrel{|\!\approx}$ on filter formulas is an* approximate entailment *if $\models_{prop} \subseteq \mathrel{|\!\approx} \subseteq \models$. We write $F \approxeq G$ if $F \mathrel{|\!\approx} G$ and $G \mathrel{|\!\approx} F$.*

We generalise Algorithm 1 to approximate entailments by replacing any use of $\models$ by $\mathrel{|\!\approx}$, and by allowing any representation function $\mathsf{rep} : \mathcal{F}_k \to \mathcal{F}_k$ where $F \approxeq \mathsf{rep}(F)$ and $F \approxeq G$ implies $\mathsf{rep}(F) = \mathsf{rep}(G)$. Definition 4 and Algorithm 2 can likewise be generalised by using filters computed by the approximate algorithm. The main results of Sections 3 and 4 still hold in this generalised setting.

▶ **Lemma 17.** *For any approximate entailment $\mathrel{|\!\approx}$, Algorithm 1 terminates within the bounds of Theorem 11. If P' is an admissible rewriting of P based on $\mathrel{|\!\approx}$, and $p(\boldsymbol{c})$ is a fact with $p \in \mathbf{P}_{\mathsf{out}}$, then $P, \mathcal{D} \models p(\boldsymbol{c})$ iff $P', \mathcal{D} \models p(\boldsymbol{c})$.*

If $\mathrel{|\!\approx}$ is decidable, Algorithm 1 can be implemented, but may still be intractable. In fact, deciding $\models_{\text{prop}}$ is still hard for *co*NP. However, weakening $\mathrel{|\!\approx}$ further to omit entailments of $\models_{\text{prop}}$ may impair termination, since propositionally equivalent formulas may have distinct

representatives. To obtain a tractable procedure, we further modify Algorithm 1 so that the formulas $\mathsf{flt}(p)$ can only be *conjunctions* of filter atoms, $\top$, or $\bot$. Lines L7 and L8 in the algorithm are now replaced by

$$\mathsf{flt}(b) := \bigwedge \{A \in \mathcal{A} \mid \iota_{b(\boldsymbol{y})}(\mathsf{flt}(b)) \vee G \mathrel{\mathrel{|}\!\approx} \iota_{b(\boldsymbol{y})}(A)\} \tag{17}$$

where $\mathcal{A} = \{\bot\} \cup \mathbf{F}[\mathsf{ar}(b)]$, and we assume that conjunctions are represented as subsets of $\mathcal{A}$ with $\bigwedge \emptyset = \top$. We refer to the modified algorithm as *conjunctive approximate static filtering* (CASF). The complexity depends on the choice of $\mathrel{|}\!\approx$ and the size of $\mathbf{F}[\mathsf{ar}(b)]$, but the algorithm is correct in all cases. This follows from the observation that the formulas $\mathsf{flt}(b)$ as computed in CASF are logical consequences (w.r.t. $\models$) of those computed in Algorithm 1, i.e., filters are more permissive.

▶ **Theorem 18.** *If P' is an admissible rewriting of P for the filter formulas computed in CASF, and $p(\boldsymbol{c})$ is a fact with $p \in \mathbf{P}_{\mathsf{out}}$, then $P, \mathcal{D} \models p(\boldsymbol{c})$ iff $P', \mathcal{D} \models p(\boldsymbol{c})$.*

For polynomial runtime, we need to restrict $\mathrel{|}\!\approx$ so that the entailment in (17) can be decided in P. We consider a finite set $\mathcal{T}$ of Datalog rules that use only filter predicates in head and body. $\mathcal{T}$ is a *Horn axiomatisation* for $\mathrel{|}\!\approx$ if $F \mathrel{|}\!\approx G$ holds for filter formulas $F, G \in \mathcal{F}_k$ exactly if G is a logical consequence of $F \cup \mathcal{T}$ (considered as a predicate logic theory over the domain of positional markers $\mathbf{N}_k$). Moreover, $\mathcal{T}$ is a *linear axiomatisation* if rules in $\mathcal{T}$ have exactly one body atom.

▶ **Theorem 19.** *Let $\mathbf{F}$ be a set of filter predicates with bounded arity, and let $\mathcal{T}$ be a (fixed) Horn approximation of $\mathrel{|}\!\approx$. Then CASF can be executed in polynomial time over a program P in either of the following cases:*

1. *$\mathcal{T}$ is a linear approximation, or*
2. *the filter expressions $G_{\mathbf{F}}$ in P do not contain $\vee$.*

Linear rules as in the first case in Theorem 19 suffice to model basic hierarchies of filter conditions, but the power of Horn logic is required to axiomatise typical binary filters such as order relations $\leq$ [33].

▶ **Example 20.** For a set of natural numbers $N \subseteq \mathbb{N}$, consider the (possibly infinite) theory

$$x \leq \mathtt{c} \leftarrow x \doteq \mathtt{c} \qquad \mathtt{c} \in N \tag{18}$$

$$x \leq \mathtt{c} \leftarrow y \leq \mathtt{c} \wedge y \doteq x{+}\mathtt{d} \qquad \mathtt{c}, \mathtt{d} \in N \tag{19}$$

$$x \leq \mathtt{c} \leftarrow x \leq \mathtt{d} \qquad \mathtt{c}, \mathtt{d} \in N, \mathtt{c} > \mathtt{d} \tag{20}$$

with rules instantiated for all values of $\mathtt{c}$ and d as specified. The finite instantiation with $N = \{0, 1, 5\}$ axiomatises all filter entailments necessary for Examples 3 and 6. The relevant constants N are syntactically given in the input filters.

We observe that the cases in Theorem 19 cannot be combined without loosing tractability:

▶ **Proposition 21.** *There is a Horn approximation $\mathcal{T}$ of $\mathrel{|}\!\approx$, such that deciding $G \mathrel{|}\!\approx A$ for a filter formula G and an atom A is hard for co*NP.

Tractable static filtering for real-world data

Finally, we investigate the impact of (tractable) static filtering for reasoning over real-world data. We implement conjunctive approximate static filtering (CASF) for linear orders and instantiations of the rules of Example 20 for all natural numbers in a program P. Moreover,

$$tc(x,y) \leftarrow p(x,y) \qquad tc(x,y) \leftarrow p(x,y) \wedge x \doteq \mathtt{a}$$
$$tc(x,z) \leftarrow tc(x,y) \wedge p(y,z) \qquad tc(x,z) \leftarrow tc(x,y) \wedge p(y,z)$$
$$out(y) \leftarrow tc(x,y) \wedge x \doteq \mathtt{a} \qquad out(y) \leftarrow tc(x,y)$$

Figure 1 Template programs for transitive closure over some EDB predicate $p \in \mathbf{P}$; some filter predicate $x \doteq a \in \mathbf{F}$ with constant $\mathtt{a}$ is applied to compute the output predicate $out \in \mathbf{P}_{\mathsf{out}}$; original program (left) and rewritten program by tractable static filtering (right).

Property p	Property name	#Facts	Entity a
P2652	partnership with	6,638	Q180 (Wikimedia Foundation)
P530	diplomatic relation	7,290	Q33 (Finland)
P1327	partner in business or sport	27,716	Q1203 (John Lennon)
P197	adjacent station	266,608	Q219867 (London King's Cross)
P47	shares border with	927,553	Q33 (Finland)

Figure 2 Table of used Wikidata properties used in the evaluation, i.e., the programs in Figure 1 are instantiated with properties p and entities $\mathtt{a}$; #Facts is the number of facts for property p.

we treat EDB predicates as filter predicates. We rewrite programs for transitive closure over Wikidata properties via CASF, and we compare the runtime of the original programs with the rewritten ones as well as the runtime for static filtering. In particular, we show that

1. static filtering can improve the performance of modern rule systems by orders of magnitude,
2. the simplifications in Section 5 are general enough to obtain these improvements, and
3. the time necessary for applying tractable static filtering is negligible.

As a typical examples for recursive programs, we use programs for computing the transitive closure of a predicate, and we add an output predicate *out* and a filter $x \doteq \mathtt{a}$ for a constant $\mathtt{a}$. We use a template for transitive closure (see Figure 1), which we instantiated for different EDB predicates $p(x,y)$. We extract these predicates from Wikidata properties (see Figure 2). As rule systems, we considered Soufflé v2.5 [20], Nemo v0.8.1 [18], Clingo v5.8.0 [13], and DLV v2.1.2 [2]. We have adopted the programs and inputs to the capabilities of the rule systems: Soufflé and Nemo received facts as CSV files, while Gringo and DLV received them as a list of facts. We applied a timeout at 5min. Our measurements are performed on a regular notebook (Linux; AMD Ryzen 7 PRO 5850U; 16 GiB RAM).

Figure 3 shows the runtimes for Programs 1 instantiated with the properties of Table 2. For each property, we run each system with the original program and the rewritten one. In all cases, the rewritten programs require significantly less time – note the log-scale of the plot. For the properties where the systems could finish for the original program, static filtering provides a speed-up in the order of magnitude (from 6.6 times for P1203 and Clingo to 30 times for P2652 and DLV). Moreover, static filtering enables all systems to finish for huge properties with up to $1,000,000$ facts within seconds, while all system reached the timeout for the original program there. Finally, we observe that static filtering can be done in a few milliseconds, and it is independent of the number of facts for the underlying property.

Our experiments show that (tractable) static filtering can improve the performance of rule systems significantly. However, evaluating a static optimisation method empirically has its limitations. Static optimization, by design, works on inputs that are not manually optimized yet. While such programs are common in practice, there is no grounds for assuming anything

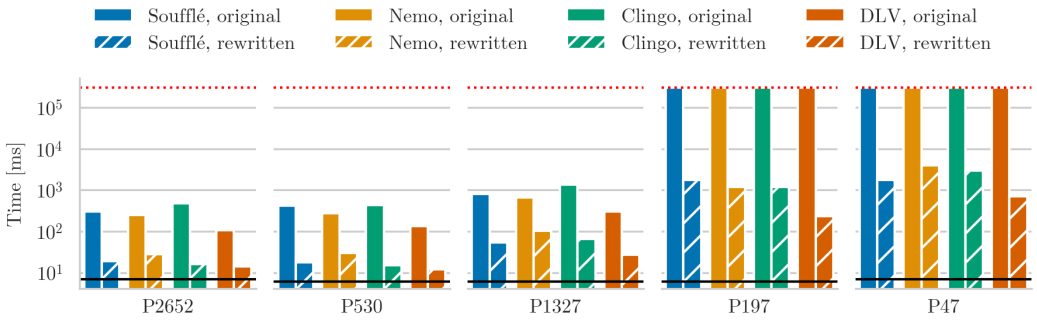

Figure 3 Runtimes (median of five runs) for transitive closure programs for different properties and rule systems; solid bars show runtime for original programs; hatched bars show runtime for rewritten programs; runtime of static filtering (black, solid lines); timeout (red, dotted lines) at 5min.

about how common this is. Existing public program collections are not representative here either, since most of these published programs have been carefully optimised by experts – exactly the type of manual work that static optimisations try to automate.

6 Incorporating Nonmonotonic Negation

In this section, we extend static filtering for rules with negation. Datalog is often extended with *stratified negation* [1]. The *stable model semantics* as used in Answer Set Programming (ASP) [7, 14] is a popular way to generalise this semantics to arbitrary rules with negation. Both cases can benefit from static filtering; especially for ASP, where a polynomial reduction in the grounding size can lead to an exponential performance advantage in solving.

Indeed, static optimisation in ASP is an important active topic of research. Existing approaches include rewritings based on tree decompositions [4, 8], projection [17], rule subsumption and shifting [10], and magic sets [9, 15] – but we are not aware of any work that resembles static filtering. Indeed, Table 1 indicates that leading ASP engines do not implement such optimisations even for basic filters of the form $\_ \doteq \mathsf{c}$. The recent tool *ngo*, maintained by the Clingo developers, likewise implements many known optimisations, but no static filtering.[4] When running *ngo* with all optimisations enabled on Example 1, it merely rewrites Rule 3 to $\mathit{out}(\mathsf{b}) \leftarrow p(x_1, \ldots, x_\ell, \mathsf{b})$, which, expectably, does not have a notable impact on the runtime of any of the systems tested.

Rules with negation

A *negated atom* is an expression $\mathit{not}\, p(\boldsymbol{t})$ with $p \in \mathbf{P}$ and $|\boldsymbol{t}| = \mathsf{ar}(p)$. A *normal rule* ρ is a formula $H \leftarrow B \wedge B^-$, where the head H is an atom, B is a conjunction of atoms, and B^- is a conjunction of negated non-filter atoms, such that every $v \in \mathsf{var}(\rho)$ occurs in some atom $p(\boldsymbol{x}) \in B$ (*safety*). Negated filter atoms are not needed: we can express them by introducing fresh filter predicates that are interpreted by the complemented relation. Analogously to Datalog, a *normal rule with generalised filter expressions* has the form $H \leftarrow B_{\bar{\mathbf{F}}} \wedge B^-_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}}$ with H a head atom, $B_{\bar{\mathbf{F}}}$ a conjunction of non-filter atoms, and $G_{\mathbf{F}} \in \mathbf{G}$ a positive boolean

[4] `https://potassco.org/ngo/ngo.html`

combination of filter atoms. In this section, all rules and programs may include negated atoms and generalised filters (we omit *normal*). The normal form without repeated variables per (negated or non-negated) atom is defined as for Datalog.

Stable models

For a program P and database $\mathcal{D}$, let $\mathsf{gr}(P) = \{\rho\sigma \mid \rho \in P, \sigma\colon \mathbf{V} \to \mathbf{C}\}$ be its *grounding*. For a set of facts $\mathcal{A}$, the Datalog program $\mathsf{gr}(P)^{\mathcal{A}} = \{H \leftarrow B \mid H \leftarrow B \wedge B^- \in \mathsf{gr}(P), B^- \cap \mathcal{A} = \emptyset\}$ is the *reduct*. $\mathcal{A}$ is a *stable model* of P and $\mathcal{D}$ if $\mathcal{A}$ is the model of $\mathsf{gr}(P)^{\mathcal{A}}$ and $\mathcal{D}$. We write $\mathsf{sm}(P, \mathcal{D})$ for the set of all such stable models.

Stratified negation

Even when using stable models, we check if programs are (partly) stratified, so as to tighten some filters. Let G_P be the graph with the IDB predicates of P as its vertices, and, for every rule $q(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge B^-_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}} \in P$, a positive edge $p \to_+ q$ for all IDBs p in $B_{\bar{\mathbf{F}}}$, and a negative edge $p \to_- q$ for all IDBs p in $B^-_{\bar{\mathbf{F}}}$. The *stratifiable predicates* $\mathbf{P}_{\mathsf{str}}$ are the vertices p of G_P such that there is no cycle C in G_P such that C a negative edge and p is reachable from C.

Static filtering for stable models

Importantly, facts that do not contribute directly to the output can still have an impact on stable models. For example, consider a program P with a stable model $\mathcal{A}$ that contains some $q(\boldsymbol{c}) \in \mathcal{A}$; then adding a rule $p(\boldsymbol{x}) \leftarrow q(\boldsymbol{x}) \wedge \mathit{not}\, p(\boldsymbol{x})$ for a fresh p means that $\mathcal{A}$ is no longer stable. Hence, static filtering must not filter facts $p(\boldsymbol{c})$ relevant to some $\mathit{not}\, p(\boldsymbol{x})$.

We therefore define more general initial filters for negated predicates. For a rule $\rho = h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge B^-_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}}$ with $b(\boldsymbol{y}) \in B^-_{\bar{\mathbf{F}}}$, let $M_{b(\boldsymbol{y})} = \bigwedge\{F \in \mathcal{F}_{\mathsf{ar}(b)} \mid G_{\mathbf{F}} \models \iota_{b(\boldsymbol{y})}(F)\}$, and let $N^p_\rho = \bigvee\{M_{p(\boldsymbol{y})} \mid p(\boldsymbol{y}) \in B^-_{\bar{\mathbf{F}}}\}$ with $\bigvee \emptyset = \bot$. To obtain a procedure for a program P, we can now modify Algorithm 1 so that the formulas $\mathsf{flt}(p)$ are initialised in line L2 with

$$\mathsf{flt}(p) \coloneqq \begin{cases} \mathsf{rep}(\top) & \text{if } p \in \mathbf{P}_{\mathsf{out}} \\ \mathsf{rep}(\bigvee\{N^p_\rho \mid \rho \in P\}) & \text{if } p \notin \mathbf{P}_{\mathsf{str}} \\ \mathsf{rep}(\bot) & \text{otherwise.} \end{cases} \tag{21}$$

Note that $\mathsf{flt}(p) \equiv \bot$ for $p \notin \mathbf{P}_{\mathsf{str}}$ if p never occurs in a negated atom. Predicates $p \notin \mathbf{P}_{\mathsf{str}}$ can be initialised with $\mathsf{rep}(\bot)$ as before, but we have to consider them in the iterative generalisation: we modify line L5 to loop over all $b(\boldsymbol{y})$ with $b(\boldsymbol{y}) \in B_{\bar{\mathbf{F}}}$ or $\mathit{not}\, b(\boldsymbol{y}) \in B^-_{\bar{\mathbf{F}}}$ for IDB predicate b.

Our remaining definitions require only minimal adaptations. For a rule $\rho = h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge B^-_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}}$, a filter formula ψ is admissible for ρ if ψ is admissible for $h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge G_{\mathbf{F}}$, and $h(\boldsymbol{x}) \leftarrow B_{\bar{\mathbf{F}}} \wedge B^-_{\bar{\mathbf{F}}} \wedge \psi$ is an admissible rewriting of ρ. An *admissible rewriting* of a program P is a set that contains an admissible rewriting of each rule of P. We can use Algorithm 2 unchanged as a practical way to find good admissible filters. Our main correctness results shows *visible equivalence* [19] between P and P':

▶ **Theorem 22.** *If P' is an admissible rewriting of program P for database $\mathcal{D}$, then $\mu\colon \mathcal{A} \mapsto \{p(\boldsymbol{c}) \in \mathcal{A} \mid \boldsymbol{c} \in \mathsf{flt}(p)^{\mathcal{D}}\}$ is a bijection between $\mathsf{sm}(P, \mathcal{D})$ and $\mathsf{sm}(P', \mathcal{D})$.*

In particular, since $\mathsf{flt}(p) \equiv \top$ for output predicates p, the restrictions of $\mathsf{sm}(P, \mathcal{D})$ and $\mathsf{sm}(P', \mathcal{D})$ to facts over output predicates coincide.

One can easily incorporate the ideas of Section 5 to obtain a tractable optimisation procedure for normal logic programs.

7 Related work

Comparison with the original algorithm

The filter computation of Kifer and Lozinskii [24] can be seen as a special case of our approach for a fixed choice of filter predicates (binary equalities and inequalities, possibly involving constants) and representation of filter formulas (in disjunctive normal form). Predicates such as $\_ \dot{=} \_ + 1$ in Example 2 are not considered as filters. The significance of our generalisation is witnessed by exponential increases in complexity (Section 4), but also by the ability to introduce tractable simplifications (Section 5). The general notion of *admissibility* and Algorithm 2 are also new.

Kifer and Lozinskii further include a similar method to propagate projections and remove unused predicate parameters. This rewriting is simpler than filter propagation. We have nothing to add to it but note that it is particularly effective if static filtering is applied first.

▶ **Example 23.** Propagation of projections does not lead to any simplification for Example 2, but leads to the following rules with reduced arities for the rewriting of Example 6:

$$r'(y, n) \leftarrow e(x, y) \wedge n \dot{=} 0 \wedge x \dot{=} \mathsf{a} \tag{22}$$

$$r'(z, m) \leftarrow r'(y, n) \wedge e(y, z) \wedge m \dot{=} n{+}1 \wedge m \leq 5 \tag{23}$$

$$\mathit{out}(y) \leftarrow r'(y, n) \tag{24}$$

Instead of computing the distance of quadratically many pairs x and y, only the distance from a to linearly many y is needed. In general, reducing predicate arities can have big performance advantages, as it may simplify data structures and execution plans.

Comparison with static optimization techniques

Kifer and Lozinskii have already compared their special case of static filtering to existing methods including magic sets [3, 28, 29], and they have have already observed that the approaches by Walker [34] and Gardarin et al. [12] are similar, yet less general. Subsequently, Chang et al. extended the original method of Kifer and Lozinskii to programs with stratified negation, which is less general than our extension to ASP.

Constraint pushing [21, 30] considers rules with constraint atoms, with a propagation scheme similar to Algorithm 1. However, the method might not terminate, as there can be infinitely many constraint atoms.

Zaniolo et al. introduce *pre-mappability (PreM)* as a sufficient condition for pushing filters into recursive rules, identify some classes of PreM filters, and use such filters to rewrite recursive programs with aggregates [36]. Our Algorithm 1 can be seen as a systematic method for producing pre-mappable filters ($\mathsf{flt}(p)$ are pre-mappable).

The *FGH-rule* by Wang et al. [35] defines a sufficient condition for rewriting a program using given output predicates (or queries), and our admissile rewritings satisfy this condition (which is true for any rewriting that produces the same output facts). Hence, static filtering offers a tractable method to find FGH-rule conforming rewritings.

In general, static filtering promises to work well with some rewriting techniques such as projection propagation and pre-mappability [36], and it is unlikely that it interferes negatively with static optimisations techniques, since static filtering preserves the program structure.

Comparison to magic sets and demand transformation

Magic sets [3, 28, 29] and the closely related *demand transformation* [31, 32] are static optimisation methods that also aim at reducing inferences by making some rule bodies more selective, and which may seem similar to static filtering on an intuitive level. However, in almost all cases, their outputs are very different from ours, for the following reasons:

1. Static filtering preserves the number of rules and the structure of their non-filter atoms. Magic sets and demand transformation always increase the number of rules and add rules that contain partial body joins.
2. Static filtering cannot optimise programs that do not contain filter predicates. Demand transformation can be used on purely abstract programs.
3. Static filtering uses symbolic reasoning over filters to simplify and summarise expressions, so rewritten rules may contain new derived filters. Magic sets and demand transformation foresee no mechanism for integrating any symbolic knowledge about existing EDB predicates, so rewritten rules are always based on copies of EDB atoms that are syntactic parts of the program.
4. Static filtering is idempotent (optimised programs are not rewritten further). The transformation by magic sets and demand transformation always changes the program, even if applied to its own output.
5. Static filtering makes use of recursive rules for recursively generalising filter expressions. Demand transformation in turn supports rewritings of IDB predicates that are defined by recursive rules.
6. Static filtering includes a simplification step that removes filters that have become redundant after pushing (via *admissibility*, Def. 4). Magic sets and demand transformation have no mechanism to detect possible simplifications.

Therefore, we do not see any general principle to obtain the benefits of static filtering from magic sets or demand transformation, even in special cases or with further adjustments. Rather, the methods are complementary and can be used together, where static filtering should go first since any reduction in body atoms can reduce the cost of the other transformation.

8 Conclusions

"It is folk wisdom that the right concepts are rediscovered several times" is how Kifer and Lozinskii started their conclusions [24]. In our work, we have revisited and generalised their original approach to static filtering, presented a tractable simplification, and shown how to extend its use to Datalog with stratified negation and ASP. Our framework lets implementers control which filters to push and which logical interactions to consider, and thus to avoid cases where the optimisation might cost more than it saves. Since static filtering also preserves rule and proof structures, it plays well with other optimisations and may even boost them (as in Example 23). It truly seems the "right concept" for many uses, in particular for data-oriented applications where logic programs play the role of queries over potentially large datasets. Thanks to the generality of our framework and the presented simplifications, we are confident that any rule-based system can find a sweet spot where a small amount of effort can yield decisive advantages at least in some cases.

Besides speeding up today's programs, however, we should also look for new uses ahead. One is modularisation, since re-usable logic programming libraries cannot be optimised manually for (yet unknown) usage contexts. Another is termination, for arithmetic features as shown in Example 6, but also for rule languages with function symbols or existential quantifiers. These and other directions merit further research.

References

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

2 Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In Marcello Balduccini and Tomi Janhunen, editors, *Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017. doi:10.1007/978-3-319-61660-5_19.

3 François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In Avi Silberschatz, editor, *Proc. 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15. ACM, 1986. doi:10.1145/6012.15399.

4 Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. *Fundam. Informaticae*, 177(3-4):275–296, 2020. doi:10.3233/FI-2020-1990.

5 Andreas Billig. A TRIPLE-oriented approach for integrating higher-order rules and external contexts. In Diego Calvanese and Georg Lausen, editors, *Proc. 2nd Int. Conf. on Web Reasoning and Rule Systems (RR'08*, volume 5341 of *LNCS*, pages 214–221. Springer, 2008. doi:10.1007/978-3-540-88737-9_17.

6 Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. Reasonable highly expressive query languages. In Qiang Yang and Michael Wooldridge, editors, *Proc. 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, pages 2826–2832. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/400.

7 Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

8 Francesco Calimeri, Simona Perri, and Jessica Zangari. Optimizing answer set computation via heuristic-based decomposition. *Theory Pract. Log. Program.*, 19(4):603–628, 2019. doi:10.1017/S1471068419000036.

9 Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Enhancing the magic-set method for disjunctive Datalog programs. In Bart Demoen and Vladimir Lifschitz, editors, *Proc. 20th Int. Conf. on Logic Programming (ICLP'04)*, volume 3132 of *LNCS*, pages 371–385. Springer, 2004. doi:10.1007/978-3-540-27775-0_26.

10 Thomas Eiter, Michael Fink, Hans Tompits, Patrick Traxler, and Stefan Woltran. Replacements in non-ground answer-set programming. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 340–351. AAAI Press, 2006. URL: http://www.aaai.org/Library/KR/2006/kr06-036.php.

11 Alvaro A. A. Fernandes, Maria L. Barja, Norman W. Paton, and M. Howard Williams. The formalisation of ROCK & ROLL: A deductive object-oriented database system. *Inf. Softw. Technol.*, 39(6):379–389, 1997. doi:10.1016/S0950-5849(96)00007-9.

12 Georges Gardarin, Christophe de Maindreville, and Eric Simon. Extending a relational DBMS towards a rule-based system: An approach using predicate transition nets. In Joachim W. Schmidt and Costantino Thanos, editors, *Foundations of Knowledge Base Management: Contributions from Logic, Databases, and Artificial Intelligence, Book resulting from the Xania Workshop 1985*, Topics in Information Systems, pages 131–152. Springer, 1985.

13 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

14 Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.*, 9(3/4):365–386, 1991. doi:10.1007/BF03037169.

15 Sergio Greco. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 15(2):368–385, 2003. doi:10.1109/TKDE.2003.1185840.

16 Philipp Hanisch and Markus Krötzsch. Rule rewriting revisited: A fresh look at static filtering for Datalog and ASP, 2026. arXiv:2601.05108.

17 Nicholas Hippen and Yuliya Lierler. Automatic program rewriting in non-ground answer set programs. In José Júlio Alferes and Moa Johansson, editors, *Proc. 21st Int. Symposium on Practical Aspects of Declarative Languages (PADL'19)*, volume 11372 of *LNCS*, pages 19–36. Springer, 2019. doi:10.1007/978-3-030-05998-9_2.

18 Alex Ivliev, Lukas Gerlach, Simon Meusel, Jakob Steinberg, and Markus Krötzsch. Nemo: Your friendly and versatile rule reasoning toolkit. In Pierre Marquis, Magdalena Ortiz, and Maurice Pagnucco, editors, *Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24)*, pages 743–754. IJCAI Organization, 2024. doi:10.24963/kr.2024/70.

19 Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *J. Appl. Non Class. Logics*, 16(1-2):35–86, 2006. doi:10.3166/jancl.16.35-86.

20 Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II*, volume 9780 of *LNCS*, pages 422–430. Springer, 2016. doi:10.1007/978-3-319-41540-6_23.

21 David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin, and Krishnamurthy Meenakshi. Propagating constraints in recusive deduction databases. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proc. North American Conference on Logic Programming*, pages 981–998. Citeseer, MIT Press, 1989.

22 Michael Kifer and Eliezer L. Lozinskii. Filtering data flow in deductive databases. In Giorgio Ausiello and Paolo Atzeni, editors, *Proc. 1st Int. Conf. on Database Theory (ICDT'86)*, volume 243 of *LNCS*, pages 186–202. Springer, 1986. doi:10.1007/3-540-17187-8_37.

23 Michael Kifer and Eliezer L. Lozinskii. Implementing logic programs as a database system. In *Proc. 3rd Int. Conf. on Data Engineering (ICDE'87)*, pages 375–385. IEEE Computer Society, 1987. doi:10.1109/ICDE.1987.7272403.

24 Michael Kifer and Eliezer L. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Syst.*, 15(3):385–426, 1990. doi:10.1145/88636.87121.

25 Markus Krötzsch. Modern datalog: Concepts, methods, applications. In Alessandro Artale, Meghyn Bienvenu, Yazmín Ibáñez García, and Filip Murlak, editors, *Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025)*, volume 138 of *OASIcs*. Dagstuhl Publishing, 2025. doi:10.4230/OASIcs.RW.2024/2025.7.

26 Jie Liu, Senlin Liang, Dan Ye, Jun Wei, and Tao Huang. ETL workflow analysis and verification using backwards constraint propagation. In Pascal van Eck, Jaap Gordijn, and Roel J. Wieringa, editors, *Proc. 21st Int. Conf. on Advanced Information Systems Engineering*, volume 5565 of *LNCS*, pages 455–469. Springer, 2009. doi:10.1007/978-3-642-02144-2_36.

27 David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, volume 20 of *ACM Books*, pages 3–100. ACM / Morgan & Claypool, 2018. doi:10.1145/3191315.3191317.

28 Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proc. 1990 Int. Conf. on Management of Data (SIGMOD'90)*, pages 247–258. ACM Press, 1990. doi:10.1145/93597.98734.

29 Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In Richard T. Snodgrass and Marianne Winslett, editors, *Proc. 1994 Int. Conf. on Management of Data (SIGMOD'94)*, pages 103–114. ACM Press, 1994. doi:10.1145/191839.191860.

30 Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. *J. Log. Program.*, 16(3):361–414, 1993. doi:10.1016/0743-1066(93)90048-L.

31 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient Datalog queries. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proc. 12th Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 35–44. ACM, 2010. doi:10.1145/1836089.1836094.

32 K. Tuncay Tekle and Yanhong A. Liu. Extended magic for negation: Efficient demand-driven evaluation of stratified Datalog with precise complexity guarantees. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proc. 35 Int. Conf. on Logic Programming (Technical Communications), ICLP 2019 Technical Communications*, volume 306 of *EPTCS*, pages 241–254, 2019. doi:10.4204/EPTCS.306.28.

33 Jeffrey D. Ullman and Allen Van Gelder. Efficient tests for top-down termination of logical rules. *J. ACM*, 35(2):345–373, 1988. doi:10.1145/42282.42285.

34 Adrian David Walker. *SYLLOG: A knowledge based data management system*. Department of Computer Science Courant Institute of Mathematical Sciences, 1981.

35 Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *Proc. 2022 Int. Conf. on Management of Data (SIGMOD'22)*, pages 79–93. ACM, 2022. doi:10.1145/3514221.3517827.

36 Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065, 2017. doi:10.1017/S1471068417000436.