P ≠ P

Why Some Reasoning Problems are More Tractable Than Others

Markus Krötzsch

Technische Universität Dresden, Germany

Abstract. Knowledge representation and reasoning leads to a wide range of computational problems, and it is of great interest to understand the difficulty of these problems. Today this question is mainly studied using computational complexity theory and algorithmic complexity analysis. For example, entailment in propositional Horn logic is P-complete and a specific algorithm is known that runs in linear time. Unfortunately, tight algorithmic complexity bounds are rare and often based on impractical algorithms (e.g., $O(n^{2.373})$ for transitive closure by matrix multiplication), whereas computational complexity results abound but are very coarse-grained (e.g., many P-complete problems cannot be solved in linear time).

In this invited paper, we therefore advocate another approach to gauging the difficulty of a computation: we reformulate computational problems as query answering problems, and then ask how powerful a query language is needed to solve these problems. This reduces reasoning problems to a computational model – query answering – that is supported by many efficient implementations. It is of immediate practical interest to know if a problem can be reduced to query answering in an existing database system. On the theoretical side, it allows us to distinguish problems in a more-fine grained manner than computational complexity without being specific to a particular algorithm. We provide several examples of this approach and discuss its merits and limitations.

1 Introduction

There are two main reasons for studying the complexity of computational problems. On the practical side, we want to know what is needed to solve the problem: how much time and memory will it take? On the theoretical side, we want to understand the relative difficulty of the problem as compared to others. For example, there is value in understanding that one problem is NP-complete while another is ExpTime-complete, even though we have no proof that the former is strictly easier than the latter.

The previous example also illustrates that computational complexity does often not provide insights about the worst-case complexity of algorithms implementing the problem. Such insights can rather be obtained by more detailed algorithmic analysis for specific problems. For example, it is known that the classical inference problem of computing the transitive closure of a binary relation can be solved in $O(n^{2.3729})$ [20].

Unfortunately, however, such detailed analysis is missing for most problems in knowledge representation and reasoning, and we have to be content with much coarser bounds obtained from some naive algorithm (such as the immediate $O(n^3)$ bound for transitive closure). Indeed, the efforts required for a more in-depth analysis are substantial: the study of transitive closure algorithms has a history of over four decades, and it is still unknown if the conjectured optimum of $O(n^2)$ can be reached [20]. Moreover, the best known algorithms are of limited value in practice, since the underlying view of transitive closure as matrix multiplication is rarely feasible. Even the existing results thus provide little information about the actual “difficulty” of the problem.

Computational complexity is therefore often the preferred measure for the complexity of reasoning problems (and many other problems) [17]. By abstracting worst-case estimates from specific functions to general classes of functions, we obtain broad complexity classes. One strength of this approach is that such classes are usually stable both under “simple” transformations of the problem and under modifications of the underlying computational model (the Turing Machine). This suggests that they really capture an intrinsic aspect of a computational problem. Another strength of the theory is that many (though not all) complexity classes are comparable and form a linear order. Although we often do not know if the order is strict, this provides us with a one-dimensional scale on which to specify computational difficulty.

The price to pay for this nice and clean theory is that it often tells us very little about the “real” difficulty of a problem either. Complexity classes are very broad. The class of P-complete problems, e.g., includes problems that can be solved in linear time as well as problems that are inherently quadratic, cubic, or even worse. Moreover, important complexity classes are based on non-deterministic models of computation that do not correspond to existing hardware, which is one of the reasons why it is hard to explain why exactly an NP-complete problem should be easier to implement than a PSpace-hard problem on a real (deterministic) computer.

Summing up, we have the choice between an arduous algorithmic analysis that provides detailed results for specific cases, and a more abstract complexity analysis that leads to more general but very coarse-grained classifications. In both cases, the practical significance of our insights will vary. The object of this paper is to explore a middle ground in between these two extremes.

Computation as Query Answering We propose to view computational problems as query answering problems over a database, and to explore their “difficulty” by studying how powerful a query language is needed to solve them. Indeed, query languages come in a great variety – including simple *conjunctive queries*, *regular path queries* and their extensions [18,4,6], highly expressive recursive *Datalog* queries [1], and a range of expressive fragments of Datalog [2] and higher-order logic [19] – providing us with a large space for a fine-grained classification of problems.

Databases are essentially relational structures, i.e., (hyper)graphs, and many problems admit natural representations in this format. Indeed, it is well known that the class of constraint satisfaction problems can naturally be expressed in terms of query answering [9]. In general, such a view is particularly suggestive if the original problem is based on a *set* of axioms, constraints, production rules, etc., where the actual order does not affect the meaning of the problem. This is typical for logical theories. A database, viewed as a graph, is similarly unordered, while traditional computational models require us to impose an order on the input. It is known that this has a profound impact on the computational properties of the problem: query languages are more powerful if we provide a total order on the elements of a database [10].

We will adopt a fairly simple and direct view of computational problems as graphs (databases). After this step, some query languages are sufficiently expressive to solve the original problem, while others are not. The challenge is to make this characterisation as tight as possible, i.e., to identify the "exact" point at which a query language becomes too simple to express a problem. For such results to be meaningful, we need to restrict ourselves to a relevant class of query languages that defines the space in which we want to locate computational problems. Indeed, the space of *all* query languages is so fine-grained that each problem would fall into its own class, incomparable to the others.

Our analysis will thus always be relative to a choice of query languages that provide the desired level of detail and the connection to practical tools and scenarios. We give two examples for such a choice. The first is the space of Datalog queries where the arity of intensional predicates (those that occur in rule heads) is bounded by some constant. The simplest case is *monadic Datalog*, where only unary predicates can appear in rule heads. Each increase of this arity leads to a strictly more expressive query language [2], resulting in an infinite, linear hierarchy of languages. Our second example is the space of known navigational query languages. This space consists of specific languages proposed in several papers, and we can apply our approach to relate relevant practical tasks to these practical languages.

Contributions The main contribution of this paper is to show that this approach is indeed feasible in practice. On the one hand, we find a variety of practical problems to which it is applicable. On the other hand, we show that it is indeed possible to obtain tight classifications of these problems relative to practical query languages. This requires a number of techniques for understanding the expressive power of query languages, which is an interesting side effect of this work. We argue that this view can provide fresh answers to the two original questions of computational complexity:

- It can provide relevant insights into the practical feasibility of a problem. Our chosen computational model – database query answering – is supported by many implementations. Determining whether or not a certain problem can be expressed in a practical query language can tell us something about its implementability that might be just as relevant as theoretical worst-case complexity.
- It allows us to compare the relative difficulty of problems in classes that are more fine-grained than those of complexity theory. By showing that one problem requires strictly more query power than another, we can compare problems within the framework of a relevant class of query languages. In particular, this gives us a yardstick for comparing problems within P.

In exchange for these benefits, we must give up some of the advantages of complexity theory and algorithmic analysis:

- The more fine-grained structure of query languages is tied to the fact that there are many different, mutually incomparable query languages. "Difficulty," when measured in these terms, becomes a multi-dimensional property. We can retain a one-dimensional view by restricting the freedoms we allow ourselves for choosing a query language.

- Although query answering is a practical task on which plenty of practical experience is available, it does not provide specific runtime bounds in the sense of algorithmic analysis. Even if we encode a problem using, e.g., conjunctive queries with transitive closure, we do not know which transitive closure algorithm is actually used.
- The results we obtain are not as universal or stable as those of complexity theory. Whether a problem can be solved by a query of a certain type depends on the specific representation of the problem. Syntactic structure is relevant. In compensation, we can often specify very precisely at which point a query language becomes too simple to express a problem – a similar precision can rarely be attained in complexity theory since the exact relationship between nearby complexity classes if often open.

Overall, we therefore believe that our approach is, not a replacement for, but a valuable addition to the available set of tools. Indeed, we will show a number of concrete examples where we gain specific insights that go beyond what other methods can tell us.

We use the problem of computing positive and negative entailments in propositional Horn logic as a simple example to exercise our approach. In Section 2, we introduce this example and state our main theorem: the computation of negative entailments in propositional Horn logic requires a strictly more expressive variant of Datalog than the computation of positive entailments. Towards a proof of this method, Section 3 gives formal definitions for *database*, *query*, and *retrieval problem*. We introduce Datalog and its proof trees in Section 4, derive useful characteristics of its expressivity in Section 5, and further refine this method to work with database encodings of retrieval problems in Section 6. This allows us to prove the main theorem in Section 7. In Section 8, we give an overview of several other results that have been obtained for Datalog using similar methods on various ontology languages. In Section 9, we briefly review recent query-based reasoning approaches for sub-polynomial reasoning problems, using navigational queries instead of Datalog. Section 10 sums up our results, discusses the relationship to adjacent areas, and gives some perspectives for the further development of this field.

2 Example: Propositional Horn Logic

For a concrete example, we will instantiate our framework for the case of reasoning with propositional Horn logic. Given a set of *propositional letters* **A**, the set of *propositional Horn logic rules* is defined by the following grammar:

$$\mathbf{Body} ::= \top \mid \mathbf{A} \mid \mathbf{A} \wedge \mathbf{A} \tag{1}$$

$$\mathbf{Head} ::= \bot \mid \mathbf{A} \tag{2}$$

$$\mathbf{Rule} ::= \mathbf{Body} \to \mathbf{Head} \tag{3}$$

The constants $\top$ and $\bot$ represent *true* and *false*, respectively. We restrict to binary conjunctions in rule bodies here; all our results extend to the case of *n*-ary bodies (considered as nested binary conjunctions), but we prefer the simplest possible formulation here. A *propositional Horn logic theory* is a set of propositional Horn logic rules. Logical entailment is defined as usual.

Given a Horn logic theory T we can now ask whether T is satisfiable. This is the case whenever $\bot$ is *not* derived starting from $\top$. More generally, one can also ask whether T entails some proposition b or some propositional implication $a \to b$. All of these problems are easily reducible to one another, and it is a classic result that all of them can be solved in linear time [8].

In fact, a known linear-time algorithm computes the set of all propositions that T entails to be true [8]. Interestingly, however, the situation changes if we want to compute the set of all propositions that are entailed to be false: no linear time algorithm is known for this case.[1]

What explains this unexpected asymmetry? For any given pair of propositions a and b, we can decide whether $T \models a \to b$ in linear time. We could use this to decide $T \models a \to \bot$ for all propositions a, but this would require a linear number of linear checks, i.e., quadratic time overall. Complexity theory does not provide any help either: any of the above decision problems (those with a yes-or-no answer) is P-complete [7]. On the one hand, P does not distinguish between linear and quadratic algorithms; on the other hand, the problem we are interested in is not a decision problem in the first place.

Considering that we are interested in *querying* for a set of propositions, query languages do indeed suggest themselves as a computational formalism. Here we use Datalog as one of the most well-known recursive query languages [1]. However, query languages act on databases, while our problem is given as a set of rules.

Fortunately, there are natural ways to represent Horn logic theories as databases. For this example, we use a simple encoding based on binary relations b (binary body), u (unary body) and h (head), as well as unary relations t (true) and f (false). Each rule, each propositional letter, and each of the constants $\top$ and $\bot$ are represented by a vertex in our graph. A rule $\rho : c \wedge d \to e$ is encoded by relations $b(c,\rho)$, $b(d,\rho)$, and $h(\rho,e)$, and analogously for unary rules but with u instead of b to encode the single proposition in the body. The unary relations t and f contain exactly the vertices $\top$ and $\bot$, respectively. Note that we have one explicit vertex per rule to ensure that there is no confusion between the premises of multiple rules with the same head. A Horn logic theory is now translated to a graph by translating each rule individually. The graphs of several rules can touch in the same proposition vertices, while all the rule vertices are distinct.

For the remainder of this paper, we assume that all logical theories under consideration are consistent – this alleviates us from checking the special case that all entailments are valid due to inconsistency. Now it is easy to give a Datalog query for the set of all propositions that are entailed to be true, expressed by the head predicate T:

$$t(x) \to T(x) \tag{4}$$

$$T(x) \wedge u(x,v) \wedge h(v,z) \to T(z) \tag{5}$$

$$T(x) \wedge T(y) \wedge b(x,v) \wedge b(y,v) \wedge h(v,z) \to T(z) \tag{6}$$

It is easy to verify that T contains exactly those propositions that are inferred to be true. How does this compare to the problem of finding the false propositions? The following

[1] The question if this is unavoidable and why was put forward as an open problem at the recent Dagstuhl Seminar 14201 on *Horn formulas, directed hypergraphs, lattices and closure systems*.

Datalog query computes this set in the predicate F:

$$\rightarrow I(w,w) \tag{7}$$

$$I(w,w) \wedge t(x) \rightarrow I(w,x) \tag{8}$$

$$I(w,x) \wedge u(x,v) \wedge h(v,z) \rightarrow I(w,z) \tag{9}$$

$$I(w,x) \wedge I(w,y) \wedge b(x,v) \wedge b(y,v) \wedge h(v,z) \rightarrow I(w,z) \tag{10}$$

$$I(w,x) \wedge f(x) \rightarrow F(w) \tag{11}$$

We use an auxiliary binary predicate I (implication) to compute implications between propositions. Rule (7) asserts that everything implies itself using a variable w that appears only in the head. Using such an *unsafe* rule here is not crucial: instead, one could also create several rules with non-empty bodies to find all proposition vertices.

Comparing the query in lines (4)–(6) with the query in lines (7)–(11), we can see that the only head predicate in the former is the unary T, whereas the latter also uses a binary head predicate I. It is known that Datalog with binary head predicates is strictly more expressive than monadic Datalog (where only unary head predicates are allowed) [2], so in this sense our encoding of the second problem takes up more query expressivity in this example. This might be a coincidence – maybe we just missed a simpler approach of computing false propositions – but it turns out that it is not:

Theorem 1. *To compute all propositions that are entailed to be false by a propositional Horn theory in Datalog, it is necessary to use head predicates of arity two or more.*

We prove this result in Section 7. The theorem confirms that computing false proposition is in a sense inherently more difficult than computing true propositions, when measuring difficulty relative to the expressive power of Datalog. Similar to traditional complexity classes, this insight does not provide us with specific runtime bounds: it neither shows that true propositions can be computed in linear time, nor that false propositions require at least quadratic time. Yet, it achieves what neither complexity theory nor algorithmic analysis have accomplished: to provably separate the two problems with respect to a general computational model.

Of course, the result is relative to the choice of Datalog as a model of computation. However, this covers a wide range of conceivable reasoning algorithms: almost every deterministic, polytime saturation procedure can be viewed as a syntactic variant of Datalog. Our result therefore suggests that every such procedure will face some inherent difficulties when trying to compute false propositions in linear time.

To give a proof for Theorem 1, we need to introduce various other techniques first. We begin by defining the framework of our investigation in a more rigorous way.

3 Databases, Queries, and Computational Problems

In this section, we give more precise definitions of the terms *database* and *query* that we used rather informally until now. Moreover, we specify the general type of computational problems that our approach is concerned with, and we give a canonical way of viewing these problems in terms of databases.

A database signature is a finite set Σ of relation symbols, each with a fixed arity ≥ 0. A database D over Σ consists of an *active domain* Δ^D and, for each relation symbol $r \in \Sigma$ of arity n, an n-ary relation $r^D \subseteq (\Delta^D)^n$. We require that Δ^D is countable (usually it is even finite). Other names for databases in this sense are *relational structure*, *(predicate-logic) interpretation*, and *directed hypergraph*. In particular, a database over a signature with only binary relations is the same as a directed graph with labelled edges.

In the most general sense, a query language is a decidable language (i.e., we can decide if something is a query or not) together with an interpretation that assigns to each query a function from databases to the *results* of the query over this database.

This is a very general definition. In this paper, we restrict *query* to mean *logic-based query under set semantics*. Accordingly, a query is a formula of second-order logic, where every second-order variable must be bound by a quantifier and first-order variables may occur bound or free. The number of free first-order variables is called the *arity* of the query. For simplicity, we will not consider queries that contain constants or function symbols. A query with arity 0 is called *Boolean*.

Let Q be a query of arity n, and let D be a database. A *solution* of Q is a mapping μ from free variables in Q to Δ^D, such that $D \models \mu(Q)$, i.e., the formula Q is satisfied by the database (interpretation) D when interpreting each free variable x as $\mu(x)$. In other words, we check if D is a model for "$\mu(Q)$" under the standard second-order logic semantics (the details are inessential here; we give the semantics for concrete cases later). The *result* $Q(D)$ of Q over D is the set of all solutions of Q over D. Boolean queries have at most one solution (the unique function μ with empty domain), i.e., they are either satisfied by the database or not. We tacitly identify query solutions μ with n-tuples $\langle \mu(x_1), \ldots, \mu(x_n) \rangle$, where $x_1, \ldots, x_n$ is the sequence of free variables ordered by their first occurrence in Q.

In traditional complexity theory, computational problems are often phrased as *word problems*: the input of the Turing machine is an arbitrary word over an input alphabet; the task is to decide whether this string is contained in the language or not. Input strings in this sense could be viewed as (linear) graphs, but this encoding would be impractical if we want query languages to return the actual answer to the problem. Instead, the graph we use should encode the logical relationships that are given in the input, rather than a serialization of these relations in a string. At the same time, the encoding of the problem must not make any significant changes to the formulation of the problem – in particular, it should not pre-compute any part of the solution.

Many problems in reasoning are naturally given as sets of formulae, where each formula has a well-defined term structure, i.e., formulae are defined as terms built using logical operators and signature symbols. A general framework to discuss such terms are (multi-sorted) term algebras. Instead of giving a full introduction to this field here, we give some examples.

Example 1. Formulae of propositional logic can be viewed as terms over a set of *operators*: we usually use binary operators $\wedge$, $\vee$, and $\rightarrow$; unary operator $\neg$; and sometimes also nullary operators $\top$ and $\bot$. Formulae (terms) are formed from these operators and a countably infinite set of propositional letters $\mathbf{A}$ (in algebraic terms, this set is called *generating set*) in the usual way.

Example 2. Terms of first-order logic are also terms in this sense. Here, the "operators" are given by a signature of function symbols with associated arity (we treat constants as functions of arity 0). Terms are constructed from a countably infinite set of variables $\mathbf{V}$ (the generators): every variable $x \in \mathbf{V}$ is a term; and for every n-ary function f and terms $t_1, \ldots, t_n$, the expression $f(t_1, \ldots, t_n)$ is a term.

Example 3. Formulae of first-order logic can also be viewed as terms using the well-known logical operators. In this case, the set of generators are the logical atoms, constructed using a first-order signature of predicate symbols and an underlying language of first-order terms. One can view such formulae as terms over a multi-sorted algebra, where we use multiple *sorts* (first-order terms, formulae, ...) and the arity of operators (function symbol, predicate, logical operator) is given as the signature of a function over sorts. For example, the operator $\wedge$ is of type $\mathsf{formula} \times \mathsf{formula} \to \mathsf{formula}$ while a ternary predicate is of type $\mathsf{term} \times \mathsf{term} \times \mathsf{term} \to \mathsf{formula}$. If we introduce a sort $\mathsf{variable}$, we can view quantifiers as binary operators $\mathsf{variable} \times \mathsf{formula} \to \mathsf{formula}$.

As we are mainly interested in logic here, we use *formula* (rather than *term*) to refer to logical expressions. There is usually some choice on how to capture a logical language using (multi-sorted) operators and generating sets, but in all cases formulae have a natural representation as a labelled tree structure:

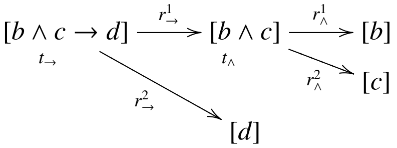
Definition 1. *Consider a set* $\mathbf{L}$ *of formulae over some signature of operators (which might include predicates and function symbols). For a formula* $\varphi \in \mathbf{L}$*, the set* $\mathsf{ST}(\varphi)$ *of* subterms *of* φ *is defined inductively as usual:* $\varphi \in \mathsf{ST}(\varphi)$ *and, if* $\varphi = o(\varphi_1, \ldots, \varphi_n)$ *for some n-ary operator o, then* $\mathsf{ST}(\varphi_i) \subseteq \mathsf{ST}(\varphi)$ *for each* $i \in \{1, \ldots, n\}$*. Moreover,* φ *is associated with a graph* $\mathsf{G}(\varphi)$ *defined as follows:*

- $\mathsf{G}(\varphi)$ *contains the vertex* $[\varphi]$*,*
- *if* φ *is of the form* $o(\varphi_1, \ldots, \varphi_n)$ *then* $[\varphi]$ *is labelled by* t_o *and, for each* $i \in \{1, \ldots, n\}$*,* $\mathsf{G}(\varphi)$ *contains an edge from* $[\varphi]$ *to* $[\varphi_i]$ *with label* r_o^i*, and* $\mathsf{G}(\varphi_i) \subseteq \mathsf{G}(\varphi)$*.*

We call a finite set $T \subseteq \mathbf{L}$ *a* theory*. The subterms of* T *are defined by* $\mathsf{ST}(T) := \bigcup_{\varphi \in T} \mathsf{ST}(\varphi)$*.* T *is associated with a graph* $\mathsf{G}(T) := \bigcup_{\varphi \in T} \mathsf{G}(\varphi)$ *over the set of vertices* $\{[\varphi] \mid \varphi \in \mathsf{ST}(T)\}$*, i.e., vertices for the same subterm are identified in* $\mathsf{G}(T)$*.*

Note that the number of vertices and edges of $\mathsf{G}(T)$ is linear in the size of T. The names we use to refer to vertices can also be of linear size, but they are only for our reference and should not be considered to be part of the database (the query languages we study do not "read" labels). Thus, the translation is linear overall.

Example 4. The propositional Horn rule $b \wedge c \to d$ is represented by the following graph structure $\mathsf{G}(b \wedge c \to d)$, where $t_\to$ and $t_\wedge$ are vertex labels:



A propositional Horn theory is translated by taking the union of the graphs obtained by translating each of its rules. For example, the translation of the theory $\{a \to b, a \to c, b \wedge c \to d\}$ can be visualized as follows, where we omit the vertex labels:

$$\begin{tikzcd}
{[d]} & {[b \wedge c \to d]} \arrow[l, "r^2_\to"'] \arrow[r, "r^1_\to"] & {[b \wedge c]} \arrow[r, "r^1_\wedge"] \arrow[dr, "r^2_\wedge"'] & {[b]} & {[a \to b]} \arrow[l, "r^2_\to"'] \arrow[r, "r^1_\to"] & {[a]} \\
 & & & {[c]} & {[a \to c]} \arrow[l, "r^2_\to"'] \arrow[ur, "r^1_\to"'] &
\end{tikzcd}$$

This provides us with a canonical representation of a broad class computational problems as graphs (and thus databases). Many problems are naturally given by sets of terms or formulae, in particular most logical reasoning tasks. Sometimes a slightly simpler encoding could be used too (e.g., the graphs in Example 4 are more "verbose" than those used for the intuitive introduction in Section 2), but results obtained for the canonical encoding usually carry over to such simplifications.

It is easy to formulate decision problems with respect to this encoding. However, our use of query languages allows us to generalise this to a broader class of *retrieval problems*, which will be the main notion of computational problem we study.

Definition 2. *Consider a set* $\mathbf{L}$ *of logical formulae. For* $n \geq 0$*, an* n-ary retrieval problem *is given by the following:*

- *a set* $\mathscr{T}$ *of theories of* $\mathbf{L}$*;*
- *a mapping* Θ *from theories* $T \in \mathscr{T}$ *to subsets of* $\mathsf{ST}(T)^n$.

An n-ary retrieval problem is solved *by an n-ary query Q if, for all theories* $T \in \mathscr{T}$*, the result* $Q(\mathsf{G}(T))$ *of Q over* $\mathsf{G}(T)$ *is exactly the set* $\Theta(T)$ *(where we identify vertices* $[\varphi]$ *with subterms* φ*).*

Decision problems are obtained as retrieval problems of arity $n = 0$. Computing all true (respectively false) propositions in Horn logic is a unary retrieval problem. Also note that we do not require Q to decide whether the given database is actually of the form $\mathsf{G}(T)$ for some theory $T \in \mathscr{T}$: the query only needs to be correct for databases that actually encode instances of our computational problem.

4 Datalog with Bounded Arity

In this paper, we mainly consider Datalog as our query language [1] and obtain a hierarchy of increasing expressivity by varying the maximal arity of head predicates. In this section, we define the basic notions that we require to work with Datalog.

Let Σ be a database signature as in Section 3. A Datalog query over Σ is based on an extended set of relation (or predicate) symbols $\Sigma' \supseteq \Sigma$. Predicates in Σ are called *extensional* (or *extensional database*, EDB), and predicates in $\Sigma' \setminus \Sigma$ are called *intensional* (or *intensional database*, IDB). We also use a fixed, countably infinite set $\mathbf{V}$ of *variables*.

A *Datalog atom* is an expression $p(x_1, \ldots, x_n)$ where $p \in \Sigma'$ is of arity n, and $x_i \in \mathbf{V}$ for $i = 1, \ldots, n$. We do not consider Datalog queries with constant symbols here. An IDB (EDB) atom is one that uses an IDB (EDB) predicate. A *Datalog rule* is a formula of the form $B_1 \wedge \ldots \wedge B_\ell \to H$ where B_i and H are Datalog atoms, and H is an IDB atom. The premise of a rule is also called its *body*, and the conclusion is called its *head*. A *Datalog query* $\langle P, g \rangle$ is a set of Datalog rules P with a *goal predicate* $g \in \Sigma' \setminus \Sigma$.

Under the logical perspective on queries of Section 3, a Datalog query corresponds to a second-order formula with IDB predicates representing second-order variables. For our purposes, however, it makes sense to define the semantics of Datalog via *proof trees*.

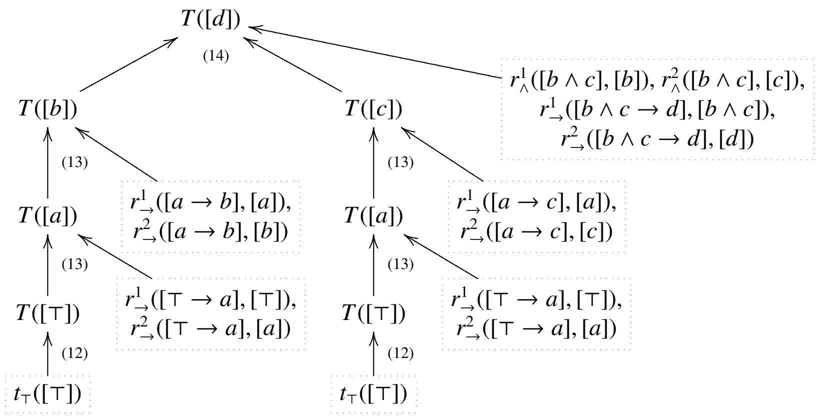

$T([d])$
(14)
$T([b])$
$T([c])$
$r^1_\land([b \land c],[b]), r^2_\land([b \land c],[c]),$
$r^1_\to([b \land c \to d],[b \land c]),$
$r^2_\to([b \land c \to d],[d])$
(13)
(13)
$T([a])$
$r^1_\to([a \to b],[a]),$
$r^2_\to([a \to b],[b])$
$T([a])$
$r^1_\to([a \to c],[a]),$
$r^2_\to([a \to c],[c])$
(13)
(13)
$T([\top])$
$r^1_\to([\top \to a],[\top]),$
$r^2_\to([\top \to a],[a])$
$T([\top])$
$r^1_\to([\top \to a],[\top]),$
$r^2_\to([\top \to a],[a])$
(12)
(12)
$t_\top([\top])$
$t_\top([\top])$

Fig. 1. Proof tree for Example 5 with leaf nodes combined in dotted boxes; labels (12)–(14) refer to the rule applied in each step

Consider a database D with active domain Δ^D. A *ground atom* for an n-ary (IDB or EDB) predicate p is an expression of the form $p(d_1, \ldots, d_n)$ where $d_1, \ldots, d_n \in \Delta^D$. A *variable assignment* μ for D is a function $\mathbf{V} \to \Delta^D$. The *ground instance* of an atom $p(x_1, \ldots, x_n)$ under μ is the ground atom $p(\mu(x_1), \ldots, \mu(x_n))$.

A *proof tree* for a Datalog query $\langle P, g\rangle$ over a database D is a structure $\langle N, E, \lambda\rangle$ where N is a finite set of nodes, $E \subseteq N \times N$ is a set of edges of a directed tree, and λ is a labelling function that assigns a ground atom to each node, such that the following holds for each node $n \in N$ with label $\lambda(n) = p(d_1, \ldots, d_k)$:

- if p is an EDB predicate, then n is a leaf node and $\langle d_1, \ldots, d_k\rangle \in p^D$;
- if p is an IDB predicate, then there is a rule $B_1 \land \ldots \land B_\ell \to H \in P$ and a variable assignment μ such that $\lambda(n) = \mu(H)$ and the set of child nodes $\{m \mid \langle n, m\rangle \in E\}$ is of the form $\{m_1, \ldots, m_\ell\}$ where $\lambda(m_i) = \mu(B_i)$ for each $i = 1, \ldots, \ell$.

A tuple $\langle d_1, \ldots, d_k\rangle$ is a solution of $\langle P, g\rangle$ over D if there is a proof tree for $\langle P, g\rangle$ over D with root label $g(d_1, \ldots, d_k)$.

Example 5. The Datalog query (4)–(6) of Section 2 can be reformulated for the canonical graphs of Section 3, as illustrated in Example 4. Note that $\top$ is a nullary operator.

$$t_\top(x) \to T(x) \tag{12}$$

$$T(x) \land r^1_\to(v, x) \land r^2_\to(v, z) \to T(z) \tag{13}$$

$$T(x) \land T(y) \land r^1_\land(w, x) \land r^2_\land(w, y) \land r^1_\to(v, w) \land r^2_\to(v, z) \to T(z) \tag{14}$$

Here, $r^1_\to$, $r^2_\to$, $r^1_\land$, $r^2_\land$, and $t_\top$ are EDB predicates from Σ, and T is the only additional IDB predicate in $\Sigma' \setminus \Sigma$. For example, rule (14) states that, whenever x and y are true, and there is a rule v of form $x \land y \to z$, then z is also true. Now consider the propositional Horn theory $\{\top \to a, a \to b, a \to c, b \land c \to d\}$, which is the same as in Example 4 but with an added implication $\top \to a$. The proof tree of $T([d])$ is shown in Fig. 1.

5 The Limits of Datalog Expressivity

To prove Theorem 1, we need to show that no monadic Datalog query can correctly compute the false propositions from a propositional Horn theory. To accomplish this, we first study the properties of retrieval problems that *can* be solved in monadic Datalog. We can then proceed to show that some of these properties are violated by (certain instances of) the logical entailment question that we are interested in.

Datalog has a number of general properties that we can try to exploit. Most obviously, Datalog is *deterministic*, or, in logical terms, it does not support disjunctive information. However, this feature is not immediately related to the arity of predicates. Another general property is that Datalog is *positive* in the sense that it only takes positive information into account during query evaluation: the absence of a structure cannot be detected in Datalog. In logical terms, this corresponds to a lack of negation; in model-theoretic terms, it corresponds to *closure of models under homomorphisms*. This feature is an important tool in our investigations:

Definition 3. *Consider databases D_1 and D_2 with active domains Δ_1^D and Δ_2^D and over the same database signature Σ. A function $\mu : \Delta_1^D \to \Delta_2^D$ is a* homomorphism *from D_1 to D_2 if, for all n-ary relations $r \in \Sigma$ and elements $d_1, \ldots, d_n \in \Delta_1^D$, we have that $\langle d_1, \ldots, d_n \rangle \in r^{D_1}$ implies $\langle \mu(d_1), \ldots, \mu(d_n) \rangle \in r^{D_2}$.*

Consider a query Q over Σ. Q is closed under homomorphisms *if, for all databases D_1 and D_2 over Σ, and every homomorphism $\mu : D_1 \to D_2$, we have $\mu(Q(D_1)) \subseteq Q(D_2)$, where $\mu(Q(D_1))$ is the set obtained by applying μ to each element in the query result $Q(D_1)$. In particular, if Q is Boolean and D_1 is a model of Q, then D_2 is a model of Q.*

It is not hard to show that Datalog is closed under homomorphisms in this sense. The utility of this observation is that it allows us to restrict attention to a much more select class of databases. Intuitively speaking, if D_1 has a homomorphism to D_2, then D_1 is less specific than D_2. We are interested in cases where this simplification does not eliminate any query results. This is captured in the following notion.

Definition 4. *Consider a database signature Σ. A set of databases $\mathscr{D}$ is a* covering *of a query Q if, for all databases D over Σ, there is a database $D' \in \mathscr{D}$ and a homomorphism $\mu : D' \to D$, such that $Q(D) = \mu(Q(D'))$.*

Intuitively speaking, a covering of a query represents every way in which the query can return a result. In most cases, a minimal covering (i.e., a covering not properly containing another covering) does not exist. Nevertheless, coverings provide a powerful proof mechanism for showing the expressive limits of a (positive) query language. The general strategy is: (1) show that every query of the language admits a covering that consists only of databases with a specific structural property, and (2) then find a retrieval problem that does not admit a covering with this property.

For example, Datalog admits natural coverings based on proof trees. The labels of the leaves of a proof tree define a "minimal" database in which the root of the proof tree can be inferred. However, this sub-database can still be much more specific than needed to obtain a correct proof tree.

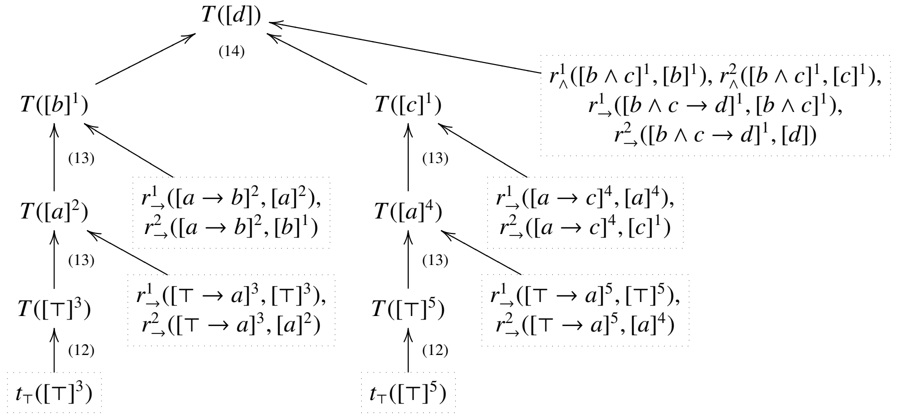

$T([d])$
(14)
$T([b]^1)$
$T([c]^1)$
$r^1_\wedge([b \wedge c]^1, [b]^1)$, $r^2_\wedge([b \wedge c]^1, [c]^1)$,
$r^1_\rightarrow([b \wedge c \rightarrow d]^1, [b \wedge c]^1)$,
$r^2_\rightarrow([b \wedge c \rightarrow d]^1, [d])$
(13)
(13)
$T([a]^2)$
$r^1_\rightarrow([a \rightarrow b]^2, [a]^2)$,
$r^2_\rightarrow([a \rightarrow b]^2, [b]^1)$
$T([a]^4)$
$r^1_\rightarrow([a \rightarrow c]^4, [a]^4)$,
$r^2_\rightarrow([a \rightarrow c]^4, [c]^1)$
(13)
(13)
$T([\top]^3)$
$r^1_\rightarrow([\top \rightarrow a]^3, [\top]^3)$,
$r^2_\rightarrow([\top \rightarrow a]^3, [a]^2)$
$T([\top]^5)$
$r^1_\rightarrow([\top \rightarrow a]^5, [\top]^5)$,
$r^2_\rightarrow([\top \rightarrow a]^5, [a]^4)$
(12)
(12)
$t_\top([\top]^3)$
$t_\top([\top]^5)$

Fig. 2. Diversified version of the proof tree in Fig. 1

Example 6. The derivation of Example 5 is possible in any database that contains the leaf labels of the according proof tree, shown in dotted boxes in Fig. 1. The subtrees for $T([b])$ and $T([c])$ both rely on the same implication $\top \rightarrow a$. Intuitively speaking, this use of a in two different branches of the proof tree is not required by the Datalog query: if we replace one of the uses of a by a', e.g., by replacing the rule $a \rightarrow c$ with $\{a' \rightarrow c, \top \rightarrow a'\}$ then we are able to derive the same result. There is a homomorphism from the modified database to the original one, mapping both a' and a to a.

This idea of renaming elements that occur only in parallel branches of a proof tree can be formalised for arbitrary Datalog queries as follows.

Definition 5. *Consider a database D, a Datalog query $\langle P, g\rangle$, and a proof tree $t = \langle N, E, \lambda\rangle$ of $\langle P, g\rangle$ on D. The* interface *of a node $n \in N$ is the set of constants from the active domain Δ^D of D that occur in its label $\lambda(n)$.*

The diversification *of the proof tree t with root node n_r is constructed recursively:*

- *for all constants $d \in \Delta^D$ that are not in the interface of n_r, introduce a fresh constant d' that has not yet been used in t or in this construction yet, and replace every occurrence of d in labels of t by d';*
- *apply the diversification procedure recursively to all subtrees of t that have a non-leaf child node of n_r as their root.*

We tacitly assume that Δ^D contains all required new constants (or we extend it accordingly). Note that the replacement of constants may not be uniform throughout the tree, i.e., several fresh constants might be introduced to replace uses of a single constant d.

Example 7. Figure 2 shows the diversified version of the proof tree for Example 5 as shown in Fig. 1. Here, we use superscript numbers to distinguish several versions of renamed vertices, where the number indicates the node in which the constant was introduced. Note that we treat expressions like $[a \rightarrow b]$ as database constants, ignoring the internal structure that their name suggests. Likewise, a diversified name like $[a \to b]^2$ is just a constant symbol. The two tree nodes originally labelled $T([a])$ now refer to distinct vertices $[a]^2$ and $[a]^4$ in the graph. The set of leaf facts shown in dotted boxes forms a diversified database with the following structure (with vertex labels $t_\to$, $t_\wedge$, and $t_\top$ omitted):

$$\begin{tikzcd}
{[d]} & {[b \wedge c \to d]^1} \arrow[l, "r^2_\to"] \arrow[d, "r^1_\to"] & & & & & \\
& {[b \wedge c]^1} \arrow[r, "r^1_\wedge"] \arrow[dr, "r^2_\wedge"'] & {[b]^1} & {[a \to b]^2} \arrow[l, "r^2_\to"] \arrow[r, "r^1_\to"] & {[a]^2} & {[\top \to a]^3} \arrow[l, "r^2_\to"] \arrow[r, "r^1_\to"] & {[\top]^3} \\
& & {[c]^1} & {[a \to c]^4} \arrow[l, "r^2_\to"] \arrow[r, "r^1_\to"] & {[a]^4} & {[\top \to a]^5} \arrow[l, "r^2_\to"] \arrow[r, "r^1_\to"] & {[\top]^5}
\end{tikzcd}$$

This database still allows for the fact $T([d])$ to be entailed, and it can be mapped by a homomorphism into the original database of Fig. 1 (partly illustrated in Example 4).

The previous example illustrates that diversified proof trees lead to diversified databases. The example considered only a single solution of the query; a more general formulation is as follows. For a Datalog query $\langle P, g\rangle$ and a database D, let $T(D, P, g)$ be the set of all proof trees for $\langle P, g\rangle$ over D. Let $\bar{T}(D, P, g)$ be a set containing one diversification for each tree in $T(D, P, g)$, where every fresh constant that was introduced during the diversification of a proof tree is distinct from all other constants throughout all diversified trees. Now the diversified database $D|_{\langle P,g\rangle}$ is defined as the union of all leaf node labels of proof trees in $\bar{T}(D, P, g)$.

Theorem 2. *Let $\mathscr{D}$ be the set of all databases over a signature Σ.*[2] *For every Datalog query $\langle P, g\rangle$ over Σ, the set $\{D|_{\langle P,g\rangle} \mid D \in \mathscr{D}\}$ is a covering for $\langle P, g\rangle$.*

This theorem can be used to show the limits of Datalog expressivity. Namely, the process of diversification is restricted by the size of the *interface* of a node in the proof tree, which in turn is closely related to the IDB arity. Therefore, a bound on the maximal IDB arity of a Datalog query affects the structure of diversifications. For example, diversifications of monadic Datalog queries are databases that can be decomposed into graphs that touch only in a single vertex, and which form a tree-like structure (technically, this is a so-called *tree decomposition* where we restrict the number of nodes that neighbouring bags may have in common). Given a query task that we conjecture to be unsolvable for monadic Datalog, we only need to find cases that do not admit such coverings. This occurs when diversification leads to databases for which strictly less answers should be returned.

For the case of retrieval problems in the sense of Definition 2, however, this approach is not quite enough yet. Indeed, a diversified database may fail to be a correct instance of the original retrieval problem. For instance, the diversified database of Example 7 does not correspond to any database that can be obtained by translating a propositional Horn theory, since two distinct vertices $[\top]^3$ and $[\top]^5$ are labelled with the unary relation $t_\top$.

[2] This is a set since we assume that every database is defined over a countable active domain.

6 Finding Diversification Coverings for Retrieval Problems

To make diversification applicable to retrieval problems, we must perform renamings in a way that guarantees that the resulting database is still (part of) a correct translation of some problem instance. Intuitively speaking, reasoning problems (and all other problems over a set of terms or formulae) are based on certain sets of atomic symbols – propositions, constants, predicate names, etc. – of which we have an unlimited supply. These sets correspond to the *generators* in a term algebra, in contrast to the *operators* of the algebra. In logic, the generators are often provided by a *signature*, but we confine our use of this term to databases and queries, and we will speak of generators and operators instead to make the distinction.

Example 8. For propositional Horn rules as defined in (1)–(3), the set $\mathbf{A}$ of propositional letters is a set of generators, whereas the constants $\top$ and $\bot$ are nullary operators. Indeed, we can freely rename propositions to create new expressions with similar properties, but there is always just a single $\top$ and $\bot$.

Applied to the idea of diversification from Section 5, this means that we can easily rename a vertex $[a]$ to $[a]'$, since the latter can be understood as a vertex $[a']$ for a new proposition a'. However, we cannot rename $[\top]$ to $[\top]'$, since it is not possible to introduce another $\top$. Additional care is needed with vertices introduced for complex formulae, since they are in a one-to-one relationship with all of their subterms. For example, a vertex $[b \wedge c]$ must always have exactly two child vertices $[b]$ and $[c]$ – we cannot rename these children independently, or even introduce multiple children (a renamed version and the original version). These informal considerations motivate the following definition:

Definition 6. *Consider a retrieval problem $\langle \mathscr{T}, \Theta\rangle$ over a language $\mathbf{L}$ of formulae as in Definition 2. Let $T \in \mathscr{T}$ be an instance of this retrieval problem, let $\langle P, g\rangle$ be a Datalog query, and let $t = \langle N, E, \lambda\rangle$ be a proof tree of $\langle P, g\rangle$ on $\mathsf{G}(T)$. The* $\mathbf{L}$-interface *of a node $n \in N$ is the set of generators of $\mathbf{L}$ that occur in the label $\lambda(n)$.*

The $\mathbf{L}$-diversification *of the proof tree t with root node n_r is constructed recursively:*

- *for all generators $d \in \Delta^D$ that are not in the $\mathbf{L}$-interface of n_r, introduce a fresh generator a' that has not yet been used in t or in this construction yet, and replace every occurrence of a in labels of t by a';*
- *apply the diversification procedure recursively to all subtrees of t that have a non-leaf child node of n_r as their root.*

Note that we replace generator symbols that are part of vertex names of the form $[\varphi]$, leading to new vertices (constants), which we assume to be added to the active domain. As before, the replacement of constants may not be uniform throughout the tree. $\mathbf{L}$-diversifications of (sets of) databases can be defined as in Section 5.

Example 9. The $\mathbf{L}$-diversification of the proof tree in Fig. 1 for $\mathbf{L}$ being the set of propositional Horn rules leads to the following diversified database:

$$\begin{tikzcd}
{[d]} & {[b^1 \land c^1 \to d]} \arrow[l, "r^2_\to"'] \arrow[d, "r^1_\to"] & & & & & \\
& {[b^1 \land c^1]} \arrow[r, "r^1_\land"] \arrow[dr, "r^2_\land"'] & {[b^1]} & {[a^2 \to b^1]} \arrow[l, "r^2_\to"'] \arrow[r, "r^1_\to"] & {[a^2]} & {[\top \to a^2]} \arrow[l, "r^2_\to"'] \arrow[r, "r^1_\to"] & {[\top]} \\
& & {[c^1]} & {[a^4 \to c^1]} \arrow[l, "r^2_\to"'] \arrow[r, "r^1_\to"] & {[a^4]} & {[\top \to a^4]} \arrow[l, "r^2_\to"'] \arrow[ur, "r^1_\to"'] &
\end{tikzcd}$$

This corresponds to the propositional Horn theory $\{\top \to a^2, \top \to a^4, a^2 \to b^1, a^4 \to c^1, b^1 \land c^1 \to d\}$, which does indeed entail that d is true while being more general than the original theory in the sense of Definition 3.

It is not hard to see that **L**-diversification is a restricted form of diversification that imposes additional restrictions for renaming vertices. Therefore, Theorem 2 still holds in this case. In addition, one can show that every **L**-diversified database is a subset of a database that is a correct encoding of some instance of the retrieval problem. For this to hold, we need to assume that the retrieval problem instances $\mathscr{T}$ contain all theories that can be obtained by renaming arbitrary occurrences of generator symbols, and by taking unions of theories in $\mathscr{T}$:

Theorem 3. *Consider a retrieval problem $\langle \mathscr{T}, \Theta \rangle$ over a language* **L**, *a theory $T \in \mathscr{T}$ and a Datalog query $\langle P, g \rangle$. Let $\mathsf{G}(T)|^{\mathbf{L}}_{\langle P,g \rangle}$ denote the* **L***-diversification of $\mathsf{G}(T)$, i.e., the database that satisfies all ground instances that occur as labels of leaf nodes of* **L***-diversified proof trees of $\langle P, g \rangle$ over $\mathsf{G}(T)$.*

If $\mathscr{T}$ is closed under unions of theories and replacements of generator symbols, then there is a theory $T|_{\langle P,g \rangle} \in \mathscr{T}$ such that $\mathsf{G}(T)|^{\mathbf{L}}_{\langle P,g \rangle} \subseteq \mathsf{G}(T|_{\langle P,g \rangle})$.

What this tells us is that **L**-diversification is a valid method for producing coverings of Datalog queries that satisfy the constraints of a certain problem encoding. It remains to use them to obtain a proof of Theorem 1.

7 Proof of Theorem 1

In this section, we bring together the tools prepared so far to complete the proof for the minimum arity of Datalog queries that solve the problem of retrieving all false propositions of a Horn logic theory.

First and foremost, we must find a suitable family of problematic cases for which an increased arity is necessary. Picking a single problem is not enough, since every finite structure can trivially be recognised by a single Datalog rule of sufficient size. We use Horn logic theories $(T_k)_{k \geq 1}$ of the form

$$T_k \coloneqq \{b_1 \land a \to \bot, b_2 \to b_1, b_3 \to b_2, \ldots, b_k \to b_{k-1}, a \to b_k\}.$$

The structure of the graph $\mathsf{G}(T_k)$ of T_k is illustrated in Fig. 3. Clearly, each T_k entails a to be false (while all other propositions might be true or false).

Now suppose for a contradiction that there is a monadic Datalog query $\langle P, g \rangle$ that solves the retrieval problem of computing false propositions from a propositional Horn theory. Thus, g has arity 1 and the answer of $\langle P, g \rangle$ on T_k is $\{[a]\}$. We show that this

$$\begin{tikzcd}
{[\bot]} & {[b_1 \wedge a \to \bot]} \arrow[l, "r_{\to}^{2}"'] \arrow[d, "r_{\to}^{1}"] & & & & & & \\
& {[b_1 \wedge a]} \arrow[r, "r_{\wedge}^{1}"] \arrow[rrr, bend right, "r_{\wedge}^{2}"'] & {[b_1]} & {[b_2 \to b_1]} \arrow[l, "r_{\to}^{2}"'] \arrow[r, "r_{\to}^{1}"] & \ldots & {[b_k \to b_{k-1}]} \arrow[l, "r_{\to}^{2}"'] \arrow[r, "r_{\to}^{1}"] & {[b_k]} & {[a \to b_k]} \arrow[l, "r_{\to}^{2}"'] \arrow[dlll, bend left, "r_{\to}^{1}"] \\
& & & & {[a]} & & &
\end{tikzcd}$$

Fig. 3. Illustration of the databases obtained from the propositional Horn theory T_k

implies that $\langle P, g\rangle$ computes some wrong answers too. Indeed, by Theorems 2 and 3, for any T_k we can find an $\mathbf{L}$-diversification T'_k for which $\langle P, g\rangle$ still returns $\{[a]\}$ as an answer. We claim that there is T_k for which this answer must be wrong, since the correct answer for any $\mathbf{L}$-diversification of T_k would be $\{\}$.

To prove this, let m be the maximal number of EDB atoms in the body of some rule in P. We set k to be $\lceil m/2 \rceil + 2$, and consider a proof tree for the solution $[a]$ of $\langle P, g\rangle$ over $\mathsf{G}(T_k)$. The root of this tree must be labelled $g([a])$.

Observation 1 All of the edges in $\mathsf{G}(T_k)$ as sketched in Fig. 3 are essential for the entailment to hold: a substructure that lacks any of these edges could always be based on some different theory that does not entail a to be false. Therefore, we can assume that each atom of $\mathsf{G}(T_k)$ occurs as a label for a leaf node in the proof tree we consider.

Observation 2 We can assume that every rule body in P, when viewed as a graph with variables as vertices and body atoms as edges, is connected and contains the variable in the head of the rule. Indeed, if a rule would contain a connected component B in the body that does not include the variable in the head, then one of three cases would apply: (1) B does not match any database of the form $\mathsf{G}(T)$ for a propositional Horn theory T, and the whole rule can be removed; (2) B matches only graphs $\mathsf{G}(T)$ for theories T that are unsatisfiable, and the whole rule can be removed (since we restrict to satisfiable theories); (3) B matches a part of $\mathsf{G}(T)$ for some satisfiable theory T, and we can change our problem formulation to include a disjoint copy of T in T_k so as to make B redundant.

From Observations 1 and 2 we obtain that any proof tree for $g([a])$ on T_k must "scan" $\mathsf{G}(T_k)$ along connected subgraphs that overlap in at least one vertex. The number of edges of these graphs is bounded by the number of body atoms in the largest body P.

Now let G_r be the graph that consists of the EDB ground atom labels of direct children of the root of the proof tree, and let $\bar{G}_r$ be $\mathsf{G}(T_k) \setminus G_r$. By our choice of k, the number of body atoms of any rule is strictly smaller than $2k - 1$, so G_r contains less than $2k - 1$ edges. Thus, $\bar{G}_r$ contains at least 3 edges of the cyclic structure of length $2k + 2$ shown in Fig. 3. Note that the edges of $\bar{G}_r$ that occur in the cycle must form a chain.

By Observation 1, the edges of $\bar{G}_r$ occur as a label in the proof tree below some child node of the root. However, all nodes have an interface of size 1 in monadic Datalog, so only the generator symbols that occur together in a single vertex can be preserved in $\bar{G}_r$ after $\mathbf{L}$-diversification as in Definition 6. It is easy to see, however, that there is no single vertex in G_r that contains enough generator symbols. For example, one of the minimal choices for $\bar{G}_r$ consists of the ground atoms $\{r^1_\rightarrow([b_i \rightarrow b_{i-1}], [b_i]), r^2_\rightarrow([b_{i+1} \rightarrow b_i], [b_i]), r^1_\rightarrow([b_{i+1} \rightarrow b_i], [b_{i+1}])\}$. In this case, G_r contains the vertices $[b_i \rightarrow b_{i-1}]$ and $[b_{i+1}]$ in the graph in its body, so it is possible that IDB child nodes in the proof tree have these vertices in their head. However, the graph that a subtree with $[b_i \rightarrow b_{i-1}]$ in its head matches after **L**-diversification has the form $\bar{G}'_r = \{r^1_\rightarrow([b_i \rightarrow b_{i-1}], [b_i]), r^2_\rightarrow([b'_{i+1} \rightarrow b_i], [b_i]), r^1_\rightarrow([b'_{i+1} \rightarrow b_i], [b'_{i+1}])\}$ where b_{i+1} was diversified to b'_{i+1}. Recall that G_r still uses the original node b_{i+1}, so the cycle does not connect in this place.

Analogously, a subtree with root vertex $[b_{i+1}]$ cannot recognize $\bar{G}_r$ either (in this case, it is b_{i-1} that can be diversified). Any chain of three edges in the cycle of Fig. 3 yields a minimal graph $\bar{G}_r$, so there are several more cases to consider; the arguments are exactly the same in each of them.

Thus, we find that the diversified graph $\mathsf{G}(T_k)'$ does not contain a cycle any more. By Theorem 2, the answer of $\langle P, g\rangle$ on $\mathsf{G}(T_k)'$ is still $\{[a]\}$. By Theorem 3 and Observation 1, however, there is a theory T'_k with $\mathsf{G}(T_k)' \subseteq \mathsf{G}(T'_k)$ which does not entail a to be false. Hence, $\langle P, g\rangle$ provides incorrect results for T'_k, which yields the required contradiction and finishes the proof.

This completes the proof of Theorem 1. It is interesting to note that the following modified theory $\hat{T}_k$ would not work for the proof: $\{b_1 \wedge a \rightarrow \bot, b_2 \wedge a \rightarrow b_1, \ldots, b_k \wedge a \rightarrow b_{k-1}, a \rightarrow b_k\}$. This modified theory still entails a to be false, but now every axiom contains the generator symbol a. This prevents **L**-diversification of a if the Datalog proof uses the nodes $[b_{i+1} \wedge a \rightarrow b_i]$ in all head atoms. Indeed, one can find a monadic Datalog query that correctly computes a to be false in all cases of this special form, and that never computes any wrong answers.

This example illustrates the crucial difference between our framework of studying retrieval problems and the traditional study of query language expressivity in database theory. Indeed, monadic Datalog cannot recognize graph structures as used in $\hat{T}_k$, yet it is possible in our setting by using the additional assumptions that the given database was obtained by translating a valid input.

8 Further Applications Based on Datalog Arity

Sections 2–7 laid out a general method of classifying the difficulty of reasoning problems based on the minimal arity of IDB predicates required to solve them. This particular instance of our approach is of special interest, due to the practical and theoretical relevance of Datalog. In this section, we give further examples where similar techniques have been used to classify more complicated reasoning tasks, and we relate this work to general works in descriptive complexity.

Many practical reasoning algorithms are based on the deterministic application of inference rules, and those that run in polynomial time can often be formulated in Datalog. A prominent example is reasoning in lightweight ontology languages. The W3C Web Ontology Language defines three such languages: OWL EL, OWL RL, and OWL QL [16,15]. OWL RL was actually designed to support ontology-based query answering with rules, but OWL EL, too, is generally implemented in this fashion [3,13,12].

Relevant reasoning tasks for these languages are the computation of instances and subclasses, respectively, of class expressions in the ontology – both problems are P-complete for OWL EL and OWL RL alike. Yet, the problems are not equivalent when comparing the required Datalog expressivity, if we consider them as retrieval problems that compute pairs of related elements:

Theorem 4 ([13]). *For OWL EL ontologies*

- *retrieving all instances-of relationships requires Datalog of IDB arity at least 3;*
- *retrieving all subclass-of relationships requires Datalog of IDB arity at least 4.*

Theorem 5 ([14]). *For OWL RL ontologies retrieving all subclass-of relationships requires Datalog of IDB arity at least 4.*

Both theorems are interesting in that they provide practically relevant problems where IDB arities of 2 or even 3 are not enough. This illustrates that our approach is meaningful beyond the (possibly rather special) case of monadic Datalog. Instead, we obtain a real hierarchy of expressivity. Nevertheless, the proof techniques used to show these results follow the same pattern that we introduced for the simpler example given herein; in particular, diversifications of proof trees play an important role [13,14].

The practical significance of these results is two-fold. On the one hand, experience shows that problems of higher minimal arity are often more difficult to implement in practice. In case of OWL EL, the arity required for class subsumption drops to 3 if a certain feature, called *nominal classes*, is omitted. And indeed, this feature has long been unsupported by reasoners [11]. Nevertheless, it is important to keep in mind that neither complexity theory nor algorithmic complexity allow us to conclude that Datalog of higher IDB arities is necessarily harder to implement, so we can only refer to practical experiences here.

On the other hand, even if it is possible in theory that efficient algorithms are unaffected by minimal IDB arities, our results impose strong syntactic restrictions on how such algorithms can be expressed in rules. This is particularly relevant for the case of OWL RL, where reasoning is traditionally implemented by rules that operate on the RDF syntax of OWL. RDF describes a graph as a set of *triples*, which can be viewed as labelled binary edges or, alternatively, as unlabelled ternary hyperedges. In either case, however, this syntax does not provide us with the 4-ary predicates needed by Theorem 5. This asserts that it is impossible to describe OWL RL reasoning using RDF-based rules[3] as implemented in many practical systems.

Finally, it is worth noting that OWL reasoning is a very natural candidate for our approach, since the official RDF syntax of OWL is already in the shape of a database in our sense. In many ways, this syntax is very similar to the canonical syntax we introduced in Section 3. The only difference is that the presence of logical operators of arbitrary arity in OWL necessitates the use of linked lists in the graph encoding. However, the use of only binary operators is a special case of this encoding, so the minimal IDB arities established here remain valid.

[3] Under the standard assumption that such rules cannot add elements to the active domain of the database during reasoning.

9 Reasoning with Navigational Query Languages

Datalog is an obvious choice for solving P-complete reasoning problems. For problems of lower complexity, however, it is more natural to consider query languages of lower data complexity. In particular, many *navigational query languages* – which view databases as a graph structure along which to navigate – are in NLogSpace for data complexity. Such languages are typically contained in linear, monadic Datalog. In a recent work, it was demonstrated how to implement this idea to solve OWL QL reasoning using the prominent SPARQL 1.1 query language [5]. Here, we give a brief overview of these results and relate them to our general framework.

Traditionally, OWL QL reasoning is often implemented by using *query rewriting*, where a reasoning task is transformed into a data access problem. This, however, is different from our setting since the ontological schema is already incorporated for computing the queries used to access the database. Bischoff et al. now introduced what they called *schema-agnostic query rewriting* where the ontology is stored in the database and not used for building the query [5]. This corresponds to our formalisation of reasoning as a retrieval problem, with the only difference that Bischoff et al. assume the standard RDF serialization of OWL ontologies rather than our canonical transformation.

SPARQL 1.1 is much weaker than Datalog, but it also supports a basic type of recursion in the form of regular expressions that can be used to specify patterns for paths in the RDF graph. This can be used for OWL QL reasoning. Bischoff et al. thus obtain fixed SPARQL 1.1 queries for retrieving all subclass-of, instance-of, and subproperty-of relationships. We omit the details here for lack of space. To provide an extremely simplified example: in a logic that only supports a binary relation subClassOf, it is possible to retrieve the entailed subclass-of relations with a single query x subClassOf* y, where * denotes the Kleene star (zero or more repetitions), and x and y are variables. It is straightforward to translate this toy example to SPARQL 1.1. Supporting all of OWL QL requires somewhat more work.

Schema-agnostic query rewriting certainly has some practical merit, allowing us to use SPARQL 1.1 database systems as OWL QL reasoners, but what does it tell us about the difficulty of the problem? For one thing, SPARQL 1.1 is already a fairly minimal navigational query language. More expressive options include nSPARQL [18], XPath [4], and other forms of nested path queries [6]. Indeed, it turns out that one feature of OWL QL – symmetric properties – cannot be supported in SPARQL 1.1 but in nSPARQL [5]. This is interesting since the feature does not otherwise add to the complexity of reasoning. In fact, one can express it easily using other features that cause no such problems for SPARQL 1.1.[4]

Nevertheless, the landscape of navigational query languages is less systematic than the neat hierarchy of Datalog of increasing IDB arity. Therefore, such results are more interesting from a practical viewpoint (What is possible on existing graph database systems?) than from a theoretical one (Is one problem harder than the other in a principled way?). However, the further development of graph query languages may lead to a more uniform landscape that provides deeper insights.

[4] In detail, the RDF graph structure p rdfs:subPropertyOf _:b . _:b owl:inverseOf p can be matched as part of regular path queries, while p rdf:type owl:SymmetricProperty cannot.

10 Outlook and Open Problems

We have presented an approach of reformulating reasoning problems in terms of query answering problems, which, to the best of our knowledge, has not been phrased in this generality before. We argued that such a viewpoint presents several benefits: its practical value is to "implement" computing tasks in the languages that are supported by database management systems; its theoretical value is to connect the difficulty of these tasks to the rich landscape of query language expressivity.

A new result established herein showed that the computation of all positive entailments of propositional Horn logic is, in a concrete technical sense, easier than the computation of all negative entailments, at least when relying on deterministic rules of inference that can be captured in Datalog. Other results we cited showed how to implement ontological reasoning in Datalog and SPARQL 1.1, explained why reasoning in OWL EL seems to become "harder" when adding certain features, and showed that schema reasoning for OWL RL cannot be described using RDF-base rules [13,14,5]. The range of these results illustrates how the proposed approach can fill a gap in our current understanding of reasoning tasks, but many problems are still open.

Our proposal has close relationships to several other fields. The relative expressiveness of query languages is a traditional topic in database theory. However, as discussed in Section 7, related results cannot be transferred naively. When using queries to solve problems, we do not require queries to work on all databases, but only on those that actually encode an instance of the problem. This distinction is rarely important when using Turing machines for computation, but it is exposed by our more fine-grained approach.

Another related field is descriptive complexity theory, where one also seeks to understand the relationship between problems solved by a query language and problems solved by a certain class of Turing machines [10,9]. The big difference to our view is that the goal in descriptive complexity is to characterize existing complexity classes using query languages. To the contrary, we are most interested in query languages that are *not* equivalent to a complexity class in this sense, so as to discover more fine-grained distinctions between computational problems. Moreover, descriptive complexity focuses on decision problems on graphs, without considering translation (and expressivity relative to a problem encoding) or retrieval problems. Nevertheless, the deep results of descriptive complexity can provide important technical insights and methods for our field as well.

The obvious next step in this field is to apply these ideas to additional computational problems. Reasoning was shown to be a fruitful area of application, but sets of terms (which we called "theories") are also the input to problems in many other fields, such as formal languages, automata theory, and graph theory. It will be interesting to see if new insights in these fields can be obtained with query-based methods. In some cases, the first step is to recognize query-based approaches as being part of this general framework. For example, Datalog has often been used as a *rule language* to solve computational problems, but it was rarely asked if simpler *query languages* could also solve the task.

A practical extension of these investigations is to explore the practical utility of these translations. Can we use existing database systems to perform complicated computations on large datasets for us? Empirical studies are needed to answer this.

Another general direction of research is to apply these ideas for the evaluation of query languages, thus turning around the original question. Indeed, problem reductions such as the ones we presented can motivate the need for a certain expressive feature in a query language. Again, this study has a practical side, since it may also guide the optimisation of query engines by providing meaningful benchmarks that are translated from other areas.

Finally, the theory we have sketched here is still in its development, and many questions remain open. Some of the methods we introduced are fairly general already, but their application in Section 7 was still rather pedestrian. It would be of great utility to flesh out more general properties, possibly graph-theoretic or algebraic, that can make it easier to see that a problem cannot be solved by means of certain queries. Another aspect that we ignored completely is the notion of problem *reductions*. Complexity theory uses many-to-one reductions to form larger complexity classes, but it is not clear which query languages can implement which kinds of reductions. Yet another possible extension would be to generalise the form of input problems we consider. While theories (as finite sets of terms) capture many problems, one could also consider more general formulations based on context-free grammars that describe problem instances.

Thus, overall, this paper is at best a starting point for further investigations in what will hopefully remain an exciting and fruitful field of study.

Acknowledgements The author wishes to thank Sebastian Rudolph for extensive discussions on the subject and valuable feedback on this paper. Moreover, thanks are due to many colleagues whose input and collaborations have contributed to this paper, in particular Stefan Bischoff, Pierre Bourhis, Tomáš Masopust, Axel Polleres, and Michaël Thomazo. The author is grateful to Marie-Laure Mugnier and Roman Kontchakov for their kind invitation to RR 2014, where this work was presented. This work was supported by the DFG in project DIAMOND (Emmy Noether grant KR 4381/1-1).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Afrati, F.N., Cosmadakis, S.S.: Expressiveness of restricted recursive queries. In: Proc. 21st Symposium on Theory of Computing Conference (STOC'89). pp. 113–126. ACM (1989)
3. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Kaelbling, L., Saffiotti, A. (eds.) Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05). pp. 364–369. Professional Book Center (2005)
4. Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. 37(4), 31 (2012)
5. Bischoff, S., Krötzsch, M., Polleres, A., Rudolph, S.: Schema-agnostic query rewriting for SPARQL 1.1. In: Proc. 13th Int. Semantic Web Conf. (ISWC'14). LNCS, Springer (2014), to appear
6. Bourhis, P., Krötzsch, M., Rudolph, S.: How to best nest regular path queries. In: Proc. 27th Int. Workshop on Description Logics (DL'14). CEUR Workshop Proceedings, CEUR-WS.org (2014), to appear
7. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys 33(3), 374–425 (2001)

8. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Logic Programming 1(3), 267–284 (1984)
9. Feder, T., Vardi, M.: The computational structure of Monotone Monadic SNP and constraint satisfaction: A study through Datalog and group theory. SIAM Journal on Computing 28(1), 57–104 (1998)
10. Grohe, M.: From polynomial time queries to graph structure theory. Commun. ACM 54(6), 104–112 (2011)
11. Kazakov, Y., Krötzsch, M., Simančík, F.: Practical reasoning with nominals in the $\mathcal{EL}$ family of description logics. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proc. 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'12). pp. 264–274. AAAI Press (2012)
12. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. Journal of Automated Reasoning 53, 1–61 (2013)
13. Krötzsch, M.: Efficient rule-based inferencing for OWL EL. In: Walsh, T. (ed.) Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11). pp. 2668–2673. AAAI Press/IJCAI (2011)
14. Krötzsch, M.: The not-so-easy task of computing class subsumptions in OWL RL. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) Proc. 11th Int. Semantic Web Conf. (ISWC'12). LNCS, vol. 7649, pp. 279–294. Springer (2012)
15. Krötzsch, M.: OWL 2 Profiles: An introduction to lightweight ontology languages. In: Eiter, T., Krennwallner, T. (eds.) Proceedings of the 8th Reasoning Web Summer School, Vienna, Austria, September 3–8 2012, LNCS, vol. 7487, pp. 112–183. Springer (2012), available at http://korrekt.org/page/OWL_2_Profiles
16. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-profiles/
17. Papadimitriou, C.H.: Computational Complexity. Addison Wesley (1994)
18. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. J. Web Semantics 8, 255–270 (2010)
19. Rudolph, S., Krötzsch, M.: Flag & check: Data access with monadically defined queries. In: Hull, R., Fan, W. (eds.) Proc. 32nd Symposium on Principles of Database Systems (PODS'13). pp. 151–162. ACM (2013)
20. Williams, V.V.: Multiplying matrices faster than Coppersmith-Winograd. In: Karloff, H.J., Pitassi, T. (eds.) Proc. 44th Symposium on Theory of Computing Conference (STOC'12). pp. 887–898. ACM (2012)