Graal: A Toolkit for Query Answering with Existential Rules

Jean-François Baget, Michel Leclère, Marie-Laure Mugnier,
Swan Rocher, and Clément Sipieter

RuleML 2015

Inria – University of Montpellier
France

Abstract. This paper presents Graal, a java toolkit dedicated to ontological query answering in the framework of existential rules. We consider knowledge bases composed of data and an ontology expressed by existential rules. The main features of Graal are the following: a basic layer that provides generic interfaces to store and query various kinds of data, forward chaining and query rewriting algorithms, structural analysis of decidability properties of a rule set, a textual format and its parser, and import of OWL 2 files. We describe in more detail the query rewriting algorithms, which rely on original techniques, and report some experiments.

1 Introduction

Existential rules, a.k.a. Datalog+, are increasingly raising interest in the knowledge representation and database communities [CGL09,BLMS11]. Indeed, they appear to be well suited for representing ontologies, particularly in the Ontology-Based Data Access framework (OBDA) [PLC$^+$08], which seeks to exploit ontological knowledge when querying data. On the one hand, existential rules extend (function-free) Horn rules, a.k.a. Datalog rules, by allowing existentially quantified variables in rule heads. This allows for asserting the existence of unknown entities, a fundamental feature for reasoning on incomplete representations of data. On the other hand, they generalize lightweight description logics used in the context of OBDA, like those underpinning the tractable profiles of the Semantic Web ontological language OWL 2.

While the issue of querying data via existential rule ontologies has been well-studied from a theoretical viewpoint, there is still a lack of software tools that would allow to improve and demonstrate the practical usability of the framework. In this paper, we present such a software, named *Graal*.[1] Graal comes in the form of a java toolkit dedicated to existential rules and oriented toward query answering tasks. The objective of Graal is to provide algorithms and utility tools that can be used as basic blocks to develop applications and carry out experimental evaluation of new solutions.

We consider knowledge bases composed of data and existential rules, as well as conjunctive queries, all seen at a logical level. The main features of Graal are the following:

[1] Graal and related tools are available at `www.github.com/graphik-team/graal`

1. a basic layer that provides generic interfaces to store and query heterogeneous data without considering the rules; these interfaces define mappings between the logical level and data stored in various systems (currently: main memory, relational databases, triple stores, graph databases);
2. 'saturation' algorithms, which apply rules on the data in a forward chaining manner; the saturated data can then be queried using the basic layer;
3. 'query rewriting' algorithms, which reformulate a conjunctive query into a set (or 'union') of conjunctive queries; the rewritten query can then be evaluated over the data using the basic layer. Furthermore, the set of rules can be partially compiled independently from any query and the rewriting process exploits this compilation to compute compact rewritings, which have a small size in practice;
4. utility tools: a format called dlgp (for 'datalog+') and its parser, decomposition of rules, structural analysis of decidability properties of a rule set, and translation of OWL 2 files into dlgp.

Graal integrates improved versions of the query rewriting algorithm PURE [KLMT15] and the rule base analyser Kiabora [LMR13]. To the best of our knowledge, the only other tool dedicated to ontological query answering with existential rules is IRIS$^{\pm}$ [GOP14], which builds on the query rewriting algorithm Nyaya.

The paper is organized as follows. Section 2 is devoted to fundamental notions on existential rules and the associated ontological query answering problem. Sections 3 to 7 present the main features of Graal as enumerated above. Since our query rewriting algorithms rely on original techniques, we present them in more detail and report experiments that demonstrate the interest of the compilation-based rewriting.

2 Fundamental Notions

We consider logical vocabularies without function symbols, hence a *term* is a variable or a constant. An *atom* is of the form $p(t_1, \ldots, t_k)$ where p is a predicate of arity k, and the t_i are terms.

The ontological query-answering problem. A *fact base* is an existentially closed conjunction of atoms. Note that variables may occur in the fact base. This allows to encode in a natural way null values in databases or blank nodes in RDF, moreover existential rules may produce new existential variables. A *conjunctive query* (CQ) is an existentially quantified conjunction of atoms (and its free variables are called *answer variables*). When it is a closed formula, it is called a *Boolean* CQ (BCQ). Hence, fact bases and BCQs have the same logical form. It is convenient to see them as sets of atoms. A *union of CQs* is a disjunction of CQs with the same answer variables.

Given existentially closed conjunctions A and B seen as sets of atoms, a *homomorphism* h from A to B is a substitution of the variables in A by terms in B such that $h(A) \subseteq B$. It is well-known that B is logically entailed by A (notation: $A \models B$) if and only if there is a homomorphism from B to A. Hence, homomorphism is a core notion for reasoning. A fact base $\mathcal{F}$ is *redundant* if there is a homomorphism from $\mathcal{F}$ to one of its strict subsets $\mathcal{F}'$ (then $\mathcal{F}$ and $\mathcal{F}'$ are equivalent).

Given a fact base $\mathcal{F}$ and a BCQ Q, the answer to Q in $\mathcal{F}$ is *positive* if $\mathcal{F} \models Q$. If Q is a non-Boolean CQ with answer variables $(x_1 \dots x_q)$, a tuple of constants $(a_1 \dots a_q)$ is an answer to Q in $\mathcal{F}$ if there is a homomorphism from Q to $\mathcal{F}$ that maps x_i to a_i for each i. In other words, $(a_1 \dots a_q)$ is an answer to Q in $\mathcal{F}$ if the answer to the BCQ obtained from Q by substituting each x_i with a_i is positive.

An *existential rule* (hereafter abbreviated to *rule*) R is a formula $\forall \boldsymbol{x} \forall \boldsymbol{y}(B[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{z}\ H[\boldsymbol{x}, \boldsymbol{z}])$ where B and H are conjunctions of atoms, respectively called the *body* and the *head* of R (as for facts and BCQs, it is convenient to see the *body* and the *head* of a rule as sets of atoms). The variables $\boldsymbol{z}$ which occur only in H are called *existential* variables. The variables $\boldsymbol{x}$, which occur in B and in H are called *frontier* variables. Since there is no ambiguity, we may omit quantifiers in rules and simply denote a rule by $B \rightarrow H$. For example, $p(x, y) \rightarrow q(x, z) \wedge s(z)$ stands for $\forall x \forall y(p(x, y) \rightarrow \exists z(q(x, z) \wedge s(z)))$.

A *fact* is a rule with an empty body, hence it is an existentially closed conjunction of atoms (and not only a ground atom). It follows that a conjunction of facts can be seen as a single fact, which explains the above definition of a fact base.

A *knowledge base* (KB) $\mathcal{K} = (\mathcal{F}, \mathcal{R})$ consists of a fact base $\mathcal{F}$ and a finite set of (existential) rules $\mathcal{R}$. The answer to a BCQ Q in $\mathcal{K}$ is *positive* if $\mathcal{K} \models Q$ (and the definition of the answer to a CQ in $\mathcal{K}$ follows). The *(ontological) query answering problem* we consider takes as input a KB $\mathcal{K} = (\mathcal{F}, \mathcal{R})$ and a CQ Q, and asks for all answers to Q in $\mathcal{K}$. This problem has long been shown undecidable for general existential rules. However, many decidable, and even tractable, classes have been exhibited.

There are two main approaches to query answering in the presence of rules. The first approach is related to forward chaining (a.k.a. *chase* in databases): it triggers the rules to build a finite representation of inferred data such that answers can be computed by evaluating the query against this representation. The second approach, first proposed for the description logic DL-Lite [CDL$^+$07], is related to backward chaining: it rewrites the query such that answers can be computed by evaluating the rewritten query against the data. We now define fundamental notions related to these approaches.

Notions related to Forward Chaining. A rule R is *applicable* to a fact base $\mathcal{F}$ if there is a homomorphism h from the body of R to $\mathcal{F}$; the result of the *application of R on $\mathcal{F}$ w.r.t. h* is $\mathcal{F} \cup h^{safe}(\mathsf{head}(R))$ where h^{safe} is a substitution of $\mathsf{head}(R)$, that replaces each x in the frontier of R with $h(x)$, and each other variable with a "fresh" variable.

Example 1. Let $\mathcal{F} = \{p(a, b), r(b)\}$ and $R = p(x, y) \wedge r(y) \rightarrow q(x, z) \wedge s(z)$. Note that z is an existential variable. R is applicable to $\mathcal{F}$ with homomorphism $\{x \mapsto a, y \mapsto b\}$. This application produces the fact $\exists z_0(q(a, z_0) \wedge s(z_0))$, where z_0 is a fresh variable. Hence, the resulting fact base is $\mathcal{F}_1 = \{p(a, b), r(b), q(a, z_0), s(z_0)\}$.

Given a BCQ Q, it holds that $\mathcal{K} \models Q$ if and only if there is a fact base $\mathcal{F}'$ obtained from $\mathcal{F}$ by a finite sequence of rule applications such that $\mathcal{F}' \models Q$. The *saturation* $\mathcal{F}^*$ of $\mathcal{F}$ with $\mathcal{R}$ is obtained from $\mathcal{F}$ by repeatedly applying rules from $\mathcal{R}$ until no new rule application can be performed. Note that $\mathcal{F}^*$ can be infinite. Given a BCQ Q, it holds that $\mathcal{K} \models Q$ if and only if $\mathcal{F}^* \models Q$. An answer to a CQ Q in $\mathcal{K}$ can thus be seen as an answer to Q in $\mathcal{F}^*$.

Notions related to Backward Chaining. Query rewriting relies on unification between the query and a rule head. Care must be taken when handling existential variables: if a term t of the query is unified with an existential variable in a rule head, all atoms in which t occurs must also be part of the unification, otherwise the result is unsound.

Example 2. Let $Q = \{q(u, v), r(v)\}$. Consider $\mathcal{F}$ and R from the previous example. Note that Q cannot be mapped by homomorphism to $\mathcal{F}_1 = \mathcal{F}^*$, hence Q has no answer in $(\mathcal{F}, \{R\})$. Assume we unify the atom $q(u, v)$ from Q with the atom $q(x, z)$ in the head of R: then, Q is rewritten into $Q_1 = \{r(v), p(u, y), r(y)\}$, which is unsound. Indeed, Q_1 can be mapped to $\mathcal{F}$ by the homomorphism $\{u \mapsto a, y \mapsto b, v \mapsto b\}$. Intuitively, the trouble is that the 'connection' between variables u and v has been lost in Q_1.

Hence, we unify a subset Q' of the query with a subset H' of a rule head. To define such a unifier, it is convenient to use a partition of the set of terms of $Q' \cup H'$. A partition π of a set of terms is said to be *admissible* if no class of π contains two constants; then a substitution σ can be obtained from π by selecting an element e_i in each class C_i of π, with priority given to constants, and setting $\sigma(t) = e_i$ for all $t \in C_i$. A *piece-unifier* of a BCQ Q with a rule $R = B \to H$ is a triple $\mu = (Q', H', \pi_\mu)$, where $Q' \subseteq Q$, $H' \subseteq H$ and π_μ is an admissible partition on the terms of $Q' \cup H'$ such that:

1. $\sigma(H') = \sigma(Q')$, where σ is a substitution obtained from π_μ;
2. if a class C_i in π_μ contains an existential variable (from H), then the other terms in C_i are variables from Q' that do not occur in $(Q \setminus Q')$.

A *piece* P of Q with respect to μ is a non-empty inclusion-minimal subset of atoms that have to be processed together, i.e., such that: for all $a \in P$ and $a' \in Q$, if a and a' share a variable unified with an existential variable of R by μ, then $a' \in P$. One can easily check that Q' is composed of pieces of Q with respect to μ (hence the name piece-unifier). The *(direct) rewriting* of Q with R with respect to μ is $\sigma(Q \setminus Q') \cup \sigma(B)$ where σ is a substitution obtained from π_μ.

Example 3. Consider again Q, R and $\mathcal{F}$. There is no piece-unifier of Q with R since, z being an existential variable, $q(u, v)$ cannot be unified with $q(x, z)$ without extending the unifier to $r(v)$, which is not possible. Let $Q_2 = \{q(u, v), q(w, v), r(u), t(w)\}$. A piece-unifier of Q_2 with R is $(\{q(u, v), q(w, v)\}, \{q(x, z)\}, \{\{u, w, x\}, \{v, z\}\})$. The corresponding rewriting is $\{r(u), t(u), p(u, y), r(y)\}$.

Given a BCQ Q, it holds that $\mathcal{K} \models Q$ if and only if there is a BCQ Q' obtained from Q by a finite sequence of (direct) query rewriting steps such that $\mathcal{F} \models Q'$. When CQs (and not only BCQs) are involved, an answer variable cannot be unified with an existential variable from a rule head. In practice, instead of making the piece-unifier definition more complex, we simply transform a CQ Q into a BCQ by adding an atom with a special predicate *ans* that contains all answer variables, which ensures that answer variables are correctly handled, and remove all atoms with predicate *ans* at the end of the rewriting process.

3 Basic Query Answering

The kernel of Graal deals with the following problems: store a fact base and answer conjunctive queries without considering the rules yet. Graal's interface considers sets of atoms (built from predicates of any non-null arity, and whose terms include variables). Answering a query is thus seen as finding homomorphisms from a set of atoms to another. However, Graal may rely upon different storage systems as well as different querying algorithms to implement these basic problems.

Storage. A set of atoms can be stored either in main memory or in secondary memory when it is very large. In main memory, the two implementations proposed are either a list of atoms (smallest memory usage, smallest cost for adding atoms), or a graph-based data structure (a better access to the data required for querying). In secondary memory, the storage systems supported by Graal can be split into three families.

- *Relational Databases* Here, an atom $p(t_1, \ldots, t_k)$ is stored as a line $(t_1, \ldots, t_k)$ in the table. Graal uses JDBC to implement relational database systems, which allows to easily plug any RDBMS that provides a JDBC driver. Graal is currently provided with a choice of MySQL, postgreSQL and SQLite.
- *Triple Stores* Here, a binary atom $p(t_1, t_2)$ is stored as a triple $(t_1 p\ t_2)$. Graal provides an implementation using Jena TDB and another using the SAIL API that allows to use Sesame triple stores, as well as any storage that also implements that API. Note that, to encode a set of arbitrary atoms into a triple store, it is first necessary to binarize these atoms.
- *Graph Databases* Here, atoms, terms and predicates are represented by nodes in a binary graph. An atom $a = p(t_1, \ldots, t_k)$ is represented by $k+1$ labeled edges: one edge labeled `predicate` between the node representing a and the node representing p, and the others labeled `term-i` between the node representing a and the node representing t_i. Currently, Graal provides two implementations of this representation. The first one uses Neo4j, the second uses the Blueprints API through which it is possible to plug in several graph database systems.

Querying. Graal comes with a generic backtrack algorithm that can compute homomorphisms regardless of the storage system used, thanks to Graal's core API. Though this algorithm does not come (yet) with any particular optimization, it allows for the quick deployment of any new storage system. Alternatively, Graal provides translations from a conjunctive query to the native querying languages of the storage mechanisms it handles: SQL queries are used to access RDBMS; SPARQL queries are used to access triple stores; and Cypher query language is used to query data encoded in Neo4j. Note that all those translations, whatever the storage system used, ensure that the same set of answers is obtained from a given CQ.

4 Saturation

Graal provides a forward chaining algorithm for existential rules, as well as several optimizations of this algorithm. The algorithm performs breadth-first saturation. The

fact base is initialized with $\mathcal{F} = \mathcal{F}_0$. Then, at each step, considering the fact base $\mathcal{F}_i$, we compute all homomorphisms from all rule bodies to $\mathcal{F}_i$. The fact base $\mathcal{F}_{i+1}$ is obtained by applying the rules following these homomorphisms on $\mathcal{F}_i$. We illustrate the saturation mechanism on the following running example.

Example 4 (Running Example). We start from a quaternary relation $project(x, y, z, w)$, which intuitively links a project identifier x, an area y, a scientific manager z and an administrative manager w. Rule R_0 decomposes this relation into binary relations $hasArea$, $hasScManager$ and $hasAdmManager$. Rules R_1 to R_3 introduce specializations of the concept $area$, namely $sensitiveArea$, itself specialized into $security$ and $innovation$. Rules R_4 and R_5 state that relations $hasScManager$ and $hasAdmManager$ are specializations of $hasManager$. Rules R_{6a} and R_{6b} state that $hasManager$ and $isManagerOf$ are inverse relations. Rule R_7 states that 'every manager manages something'. Rules R_{8a} and R_{8b} define the concept $criticalManager$ ('a critical manager is someone who manages something in a sensitive area, and reciprocally'). Finally, Rule R_9 partially defines the concept of $accreditedManager$: 'an accredited manager is necessarily someone who manages a project in a security area'.

$R_0 = project(x, y, z, w) \rightarrow hasArea(x, y) \wedge hasScManager(x, z) \wedge hasAdmManager(x, w)$
$R_1 = sensitiveArea(x) \rightarrow area(x)$
$R_2 = security(x) \rightarrow sensitiveArea(x)$
$R_3 = innovation(x) \rightarrow sensitiveArea(x)$
$R_4 = hasScManager(x, y) \rightarrow hasManager(x, y)$
$R_5 = hasAdmManager(x, y) \rightarrow hasManager(x, y)$
$R_{6a} = isManagerOf(y, x) \rightarrow hasManager(x, y)$
$R_{6b} = hasManager(y, x) \rightarrow isManagerOf(x, y)$
$R_7 = manager(x) \rightarrow isManagerOf(x, y)$
$R_{8a} = isManagerOf(x, y) \wedge hasArea(y, z) \wedge sensitiveArea(z) \rightarrow criticalManager(x)$
$R_{8b} = criticalManager(x) \rightarrow isManagerOf(x, y) \wedge hasArea(y, z) \wedge sensitiveArea(z)$
$R_9 = accreditedManager(x) \rightarrow isManagerOf(x, y) \wedge project(y, z, v, w) \wedge security(z)$

Example 5. Let $\mathcal{F} = \{accreditedManager(claire), woman(claire)\}$. The saturation at Step 1 produces the atoms $isManagerOf(claire, y_0)$, $project(y_0, z_0, v_0, w_0)$, $security(z_0)$ (by application of rule). The saturation at Step 2 produces the atom $hasManager(y_0, claire)$ (by application of rule R_{6a}), the atom $sensitiveArea(z_0)$ (by application of rule R_2), and the atoms $hasArea(y_0, z_0)$, $hasScManager(y_0, v_0)$, $hasAdmManager(y_0, w_0)$ (by application of rule R_0).

The optimizations implemented in Graal on the saturation mechanism are twofold. The first one is related to the way we detect that a rule application added new information. The default behavior of Graal is the *restricted chase* [FKMP05]: inferred atoms are not added at step $i + 1$ if there is a folding from those atoms into F_i (*i.e.,* a homomorphism from the head of the rule into F_i that preserves frontier variables according to the homomorphism used to apply the rule).

Example 6. Let $\mathcal{F} = \{manager(tom), isManagerOf(tom, project7)\}$. The application of R_7 on $\mathcal{F}$ would produce the atom $isManagerOf(tom, y_0)$. Since it folds into $\mathcal{F}$, the restricted chase does not add this atom to $\mathcal{F}$.

The second optimization is related to the selection of rules that have to be checked to generate $\mathcal{F}_{i+1}$. The *default* behavior is to check the applicability of *all* rules at each step. We may also rely upon the *graph of rule dependencies* (GRD). The nodes of this graph are the rules. There is an arc from a rule R to a rule R' if there is a piece-unifier of the body of R' (hence, seen as a query) with (the head of) R. Optionally, such an arc can be labeled with all piece-unifiers of the body of R' with R. The essential properties of the GRD are the following:

- R' depends on R (i.e., an application of R may trigger a new application of R') iff there is an arc from R to R';
- when the GRD contains no circuit (including self-loops), then the saturation halts for any fact base.

The GRD can then be used as follows: without loss of completeness, the *dependency behavior* checks for applicability at step $i+1$ solely rules that depend on rules that were successfully applied at step i; the *unifier behavior* improves the previous behavior by considering the (piece-)unifiers between a rule R_1 and a rule R_2: if R_1 was applied at step i according to a homomorphism h, and $\mu_1, \ldots, \mu_k$ are the unifiers of the body of R_2 with R_1, then any homomorphism from the body of R_2 at step $i+1$ extends a partial homomorphism $\mu_i \circ h$ that can be computed in linear time. This latter improvement not only reduces the number of rules to be checked for applicability, but also the search space for homomorphisms.

Finally, let us point out that by combining different storage methods and querying algorithms (see Section 3), different rule decompositions (see Section 6), different redundancy elimination mechanisms (restricted, core, etc...) and different rule triggering behaviors, we obtain different algorithms that can be more or less efficient for a particular application. These choices not only impact the efficiency of the saturation mechanism, but also the halting of that procedure. It is well known, for instance, that the core chase (which removes all redundancies by computing the smallest equivalent subset of atoms) halts for some instances where the restricted chase does not. Note also that the choice of rule decomposition into atomic heads may lead to the non-termination of the chase, as shown in Section 6.2 (Example 17).

5 Query Rewriting

In this section, we present the 'piece-based' rewriting technique. Two other rewriting techniques applicable to existential rules are known. The first one skolemizes the rule heads, i.e., replaces existential variables by Skolem functions (e.g., REQUIEM [Perez-Urbina et al. 2009]). The second one decomposes the unification step into two steps: factorisation of the query, and unification itself (e.g., PerfectRef [CDL$^+$07] and IRIS$^\pm$ [GOP14]). In both methods, some intermediate queries that will not yield rewritings are generated. This is avoided in piece-based rewriting.

Basic Algorithm (PURE). Given a query Q and a set of rules $\mathcal{R}$, let $\mathcal{Q}$ be the set of all rewritings that can be obtained by a sequence of direct rewritings from Q. This set is (pre-)ordered by subsumption (Q_1 subsumes Q_2 if any answer to Q_2 is an answer

to Q_1; this can be decided by a homomorphism test). When $\mathcal{Q}$ is finite, it can be seen as a UCQ. However, it is sufficient to consider $\mathcal{Q}' \subseteq \mathcal{Q}$, such that any element of $\mathcal{Q}$ is covered (i.e., subsumed) by an element of $\mathcal{Q}'$ (we say that $\mathcal{Q}'$ is a cover of $\mathcal{Q}$). All inclusion-minimal covers of $\mathcal{Q}$ have the same cardinality.

The basic query rewriting algorithm in Graal (named PURE) takes as input a CQ and a set of existential rules and outputs a minimal cover of the set of rewritings, if the set of rewritings is finite (equivalently: if there exists a UCQ-rewriting of Q). Otherwise, it may not terminate. Among the main classes of rules ensuring the existence of a UCQ rewriting for *any* CQ, we can cite *linear* rules, which generalize most DL-Lite dialects, the *sticky* family, and classes satisfying conditions expressed on a graph of *rule dependencies* (see in particular [CGL09,CGP10,BLMS11]).

The algorithm PURE starts from the set of rewritings $\mathcal{Q}_F = \{Q\}$ and proceeds in a breadth-first manner. At each step, queries from $\mathcal{Q}_F$ which have been generated at the preceding step are explored; 'exploring' a query consists of computing the set of direct rewritings of this query with all rules. Let $\mathcal{Q}_t$ be the obtained set of new queries. At the end of the step, only a minimal cover of $\mathcal{Q}_F \cup \mathcal{Q}_t$ is kept.

The computation of a minimal cover at *each* step may seem expensive, since each comparison of two queries is a homomorphism check. The point is to ensure the termination of the algorithm whenever a finite set of rewritings exists: since a set of rewritings may be infinite and still have a finite cover, a cover has to be maintained at each step (or computed after a finite number of steps). For some classes of rules, such as linear and sticky rules, this problem does not occur, and the minimal cover could be computed only once at the end of the algorithm. For a detailed presentation of the rewriting algorithm, we refer the reader to [KLMT15].

It is well known that the bottleneck of UCQ-rewriting is the size of the produced UCQ, which can be prohibitively large in practice. Graal proposes an optimized rewriting technique, presented next.

Compilation-based algorithm (PURE$_C$). We can observe that some simple rules are an obvious cause of combinatorial explosion. A typical example is that of rules describing hierarchies of concepts (seen as unary predicates), as in the following example.

Example 7. Let $R_1 \dots R_n$ be rules of the form $R_i : b_i(x) \rightarrow b_{i-1}(x)$. These rules express that the concept b_0 is specialized into concept b_1, itself specialized into b_2, etc. Let $Q = \{b_0(x_1) \dots b_0(x_k)\}$. Each atom $b_0(x_j)$ in Q is rewritten into $b_1(x_j)$, which in turn is rewritten into $b_2(x_j)$, and so on. Thus, there are $(n+1)^k$ rewritings of Q.

Now, assume that we compile the rules from the previous example into an order on predicates $b_n < b_{n-1} < ... < b_0$ and embed this order in the homomorphism notion such that a predicate b_i can be mapped to any predicate b_j such that $j \leq i$. Then, the only rewriting of Q needed to compute the answers to Q over any fact base is Q itself. We generalize this idea by compiling all rules with an atomic body as long as they do not introduce existential variables. Since the atoms in a rule may have predicates of different arity and arguments in different positions, we compute a relation on *atoms* and not only predicates. Moreover, this relation is not necessarily an order, but a *preorder* (i.e., a reflexive, transitive, but not necessarily antisymmetric relation).

A rule is said to be *compilable* if it has a single body atom, no existential variable and no constant. W.l.o.g. we also assume that a compilable rule has a single head (indeed, if the rule has no existential variable, each atom in the head forms a piece). Let $\mathcal{R}_c$ be the set of compilable rules. We compute the closure of $\mathcal{R}_c$, denoted by $\mathcal{R}_c^*$, which is the set of all rules inferred from $\mathcal{R}_c$[2], as illustrated next on the running example.

Example 8 (Running example). The compilable rules are R_0 (decomposed into 3 rules), $R_1 \dots R_5$, R_{6a}, R_{6b}. The inferred rules are the following:
$project(x, y, z, w) \rightarrow hasManager(x, z)$
$project(x, y, z, w) \rightarrow hasManager(x, w)$
$security(x) \rightarrow area(x)$
$innovation(x) \rightarrow area(x)$
$hasScManager(x, y) \rightarrow isManagerOf(y, x)$
$hasAdmManager(x, y) \rightarrow isManagerOf(y, x)$
$project(x, y, z, w) \rightarrow isManagerOf(z, x)$
$project(x, y, z, w) \rightarrow isManagerOf(w, x)$

The preorder $\preccurlyeq$ on atoms associated with $\mathcal{R}_c^*$ is as follows: given two atoms A and B, we have $A \preccurlyeq B$ if (i) $A = B$ or (ii) there is a rule $R \in \mathcal{R}_c^*$, with a homomorphism h from $body(R)$ to A such that $h(head(R)) = B$.

Example 9 (Running example). It holds that $security(u) \preccurlyeq area(u)$ by the inferred rule $security(x) \rightarrow area(x)$; and that $project(u, b, a, a) \preccurlyeq isManagerOf(a, u)$ by the rule $project(x, y, z, w) \rightarrow isManagerOf(w, x)$ and the homomorphism $h = \{x \mapsto u, y \mapsto b, z \mapsto a, w \mapsto a\}$.

Homomorphism is the fundamental notion to compute logical entailment on sets of atoms. We extend it to embed the preorder: Given sets of atoms $\mathcal{A}$ and $\mathcal{B}$, a $\preccurlyeq$-homomorphism from $\mathcal{B}$ to $\mathcal{A}$ is a substitution h from $vars(\mathcal{B})$ to $terms(\mathcal{A})$ such that for all $B \in \mathcal{B}$, there is $A \in \mathcal{A}$ with $A \preccurlyeq h(B)$. This allows to answer CQs over a KB composed of a fact base and a set of compilable rules.

Example 10 (Running example). Let $Q(x) = \{hasManager(y, x), hasArea(y, z), sensitiveArea(z)\}$, asking for managers of projects about sensitive areas. Let $\mathcal{F} = \{project(id_1, a_1, m_1, m_2), security(a_1)\}$. The answers to Q are m_1 and m_2. For m_1, we have the $\preccurlyeq$-homomorphism $h_1 = \{x \mapsto m_1, y \mapsto id_1, z \mapsto a_1\}$, with $project(id_1, a_1, m_1, m_2) \preccurlyeq hasManager(id_1, m_1)$, $project(id_1, a_1, m_1, m_2) \preccurlyeq hasArea(id_1, a_1)$ and $security(a_1) \preccurlyeq sensitiveArea(a_1)$; and similarly for m_2.

Now, let $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_e$ be a set of existential rules, where $\mathcal{R}_c$ is composed of compilable rules. $\mathcal{R}_c$ is compiled into a preorder $\preccurlyeq$ and query rewriting is performed with R_e. The preorder has to be embedded into the rewriting process, otherwise the rewriting process would not be complete, as shown in the next example.

[2] Let R_1 and R_2 be compilable rules such that $head(R_1)$ and $body(R_2)$ are unifiable by a (classical) most general unifier u. The rule *inferred* from (R_1, R_2) is $u(body(R_1)) \rightarrow u(head(R_2))$.

Example 11 (Running example). Consider again the query Q from the preceding example. There is no rewriting of Q with the non-compilable rules, whereas clearly, using the compilable Rules R_{6a}, R_0 and R_2, Q could be rewritten into $\{isManagerOf(x,y), project(y,z,z_0,w_0),\ security(z)\}$, which would then allow to obtain the rewriting $\{accreditedManager(x)\}$ with Rule R_9.

Hence, the preorder is embedded into the piece-unifier operation as well. Given a preorder $\preccurlyeq$ on atoms, a $\preccurlyeq$-piece-unifier of Q with R is a triple $\mu = (Q', H', \pi_u)$ defined similarly to a piece-unifier, with Condition 1 ($\sigma(H') = \sigma(Q')$) being replaced by: there is a surjective mapping f from $\sigma(H')$ to $\sigma(Q')$ such that, for all $A \in \sigma(H')$, we have $f(A) \preccurlyeq A$. The direct $\preccurlyeq$-rewriting of Q according to μ is $u(\mathit{body}(R)) \cup u(Q \setminus Q')$.

Example 12 (Running example). Let $Q = \{criticalManager(x), woman(x)\}$. The basic query rewriting algorithm outputs a set of 38 CQs (these CQs are pairwise incomparable w.r.t. logical entailment, hence we cannot do better if the output is a classical UCQ). The direct $\preccurlyeq$-rewriting outputs only the 3 following queries: $Q_1(x) = Q(x)$, $Q_2(x) = \{isManagerOf(x,x_1), hasArea(x_1,x_2), sensitiveArea(x_2), woman(x)\}$ and $Q_3(x) = \{accreditedManager(x), woman(x)\}$. Q_2 is a direct rewriting of Q with Rule R_{8a} and Q_3 is a direct $\preccurlyeq$-rewriting of Q_2 with Rule R_9.

The following theorem states that the process is sound and complete: given a KB $\mathcal{K} = (\mathcal{F}, \mathcal{R})$, where $\mathcal{R} = \mathcal{R}_e \cup \mathcal{R}_c$ and $\mathcal{R}_c$ is a set of compilable rules with associated preorder $\preccurlyeq$, and a BCQ Q, it holds that $\mathcal{K} \models Q$ iff there is Q' obtained by a sequence of direct $\preccurlyeq$-rewritings from Q using rules from $\mathcal{R}_e$ such that $\mathcal{F}, \mathcal{R}_c \models Q'$ (i.e., there is a $\preccurlyeq$-homomorphism from Q' to $\mathcal{F}$). For more details, the reader is referred to [KLM15].

Graal's optimized rewriting algorithm (PURE$_C$) is composed of two steps: (1) it partitions the given rule set $\mathcal{R}$ into $\mathcal{R}_e$ and $\mathcal{R}_c$, computes $\mathcal{R}_c^*$ and encodes it into a preorder $\preccurlyeq$; (2) given Q, $\mathcal{R}_e$ and $\preccurlyeq$, it outputs a minimal cover of the set of $\preccurlyeq$-rewritings (with the notion of cover being defined with respect to $\preccurlyeq$-homomorphism instead of homomorphism). Since Step 1 is independent from any query, it can be perfomed independently from Step 2. Hence, the algorithm also accepts as input $\mathcal{R}_e$, $\mathcal{R}_c^*$ and Q.

Query evaluation Let $\mathcal{Q}$ be the result of the optimized rewriting algorithm: $\mathcal{Q}$ can be seen as a 'pivotal' representation, in the sense that it can be transformed into different kinds of queries, depending on the type of data storage and the applicative context. Obviously, it can be directly evaluated with an adequate implementation of $\preccurlyeq$-homomorphism in the case the data can be loaded in main memory.[3]

Otherwise, the set $\mathcal{Q} \cup \mathcal{R}_c$ can be straightforwardly translated into a Datalog query, as illustrated in the next example, and passed to a Datalog engine.

Example 13 (Running example). From $\mathcal{Q} = \{Q_1(x), Q_2(x), Q_3(x)\}$ (see the preceding example), we build 3 Datalog rules with head $ans(x)$ (where ans is the answer predicate). E.g., from $Q_1(x)$, we obtain `ans(x):-criticalManager(x),woman(x).` The Datalog query is composed of these 3 rules and compilable rules from $\mathcal{R}_c$.

[3] The $\preccurlyeq$-homomorphism is not available yet as a standalone querying operation in the current version of Graal.

A mixed approach can be adopted with $\mathcal{R}_c$ being used to saturate the data, and $\mathcal{Q}$ being evaluated over the 'semi-saturated' data. One may even assume that all information that could be inferred by compilable rules is already present in the data, and delegate the encoding of this information to the database manager. In particular, if $\mathcal{R}_c$ is composed solely of hierarchical rules and the data are stored in a RDBMS, semantic index techniques allow to effectively avoid the computation of saturation [RC12].

When partial saturation of the data is not feasible, $\mathcal{Q}$ may also be *unfolded* into a set of CQs (i.e., a UCQ) $\mathcal{Q}'$: $\mathcal{Q}'$ is obtained from $\mathcal{Q}$ by adding, for each $Q \in \mathcal{Q}$, all Q' such that $Q' \preccurlyeq Q$ (then computing a cover). We have experimentally checked that it is more efficient to unfold $\mathcal{Q}$ than to directly compute $\mathcal{Q}'$.

Example 14 (Running example). Queries Q_1 and Q_3 are invariant by unfolding; Q_2 is unfolded into $6 \times 2 \times 3 = 36$ queries. All queries are incomparable, hence $|\mathcal{Q}'| = 38$.

Experiments We synthetize here experimental results that demonstrate the interest of compilation-based rewriting. Due to space requirements, we cannot provide the detailed results. Since benchmarks dedicated to existential rules are not available yet, we considered rule bases obtained by translation from description logics (DLs). We first carried out experiments about the query rewriting step itself. For these experiments, we considered a widely used benchmark, introduced in [PHM09], composed of DL-Lite$_\mathcal{R}$ ontologies, namely ADOLENA, STOCKEXCHANGE, UNIVERSITY and VICODI. Additionally, we considered very large DL-Lite$_\mathcal{R}$ ontologies proposed in [TSCS13], which respectively contain more than 53k and 34k rules, with 54% and 64% of compilable rules. Each ontology is provided with 5 handcrafted queries. We first evaluated the impact of rule compilation on the rewriting process, w.r.t. the rewriting size and runtime respectively. We found a huge gap between the sizes of the output; the pivotal UCQ is often restricted to a single CQ even when the classical UCQ has thousands of CQs (up to more than 30000 CQs in a case where the pivotal UCQ contains 1 CQ). Unsurprisingly, the results on the query rewriting runtimes lead to similar observations. We found that PURE$_C$ (without or with unfolding) scales well on the large ontologies. We also compared to other query rewriting tools, namely Nyaya (which was the only other tool processing existential rules, before the recent release of IRIS$\pm$), as well as some well-known DL tools. We emphasize that these DL tools exploit the particularities of DL-Lite, specially the most recent ones, namely tw-rewriting [RMKZ13] (part of the Ontop OBDA system) and Rapid [CTS11], whereas Graal and Nyaya are designed for general existential rules. Globally, PURE$_C$ behaves similarly to the fastest tools, Rapid and tw-rewriting. If we restrict the comparaison to classical UCQ output, the fastest tools are undeniably tw-rewriting and Rapid, followed by PURE$_C$ with unfolding.

We carried out additional experiments to compare the evaluation of the classical UCQ rewriting on data with the evaluation of the pivotal UCQ on data semi-saturated by compilable rules. For these experiments, we used the DL benchmark LUBM$^\exists_{20}$ proposed in [LSTW13], which comes with a data generator. This benchmark is a modification of the well-known benchmark LUBM introduced in [GPH05] (and provided with 14 queries). In particular, it yields more rules with existential variables and adds 6 challenging queries. We consider two fact bases (stored in an RBDMS) of 151 MB (10 universities) and 3266 MB (200 universities). In both cases, the ratio between the

initial base and the semi-saturated base is rather small (approx. 1.22). Note that the semi-saturation step is independent from any query, hence it can be computed only once as a preprocessing step (for information, it took 41 seconds and 15 minutes respectively). We rewrote the 20 queries associated with LUBM and LUBM$^{\exists}_{20}$. Results about the rewriting step itself confirmed the conclusions of the first experiments. Table 1 reports the evaluation runtime for each query ('UCQ': evaluation of the classical UCQ on the initial database; 'Pivotal': evaluation of the pivotal query on the semi-saturated database; 'Rew TO' and 'Ans TO': 30 minutes timeout in the rewriting step and in the evaluation step resp.; 'SQL Err.': query too large for the RDBMS). We can see that the pivotal UCQ is evaluated much more efficiently than the classical UCQ (which could even not be produced or passed to the RDBMS in several cases). Note that, despite the pivotal rewriting of q_{18} is a single CQ, it could not be evaluated, even on the smaller fact base, because it requires a large number of joins.

# univ.	Rew.	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10
10	UCQ	0.62	1.09	0.62	0.91	0.64	13.99	0.74	4.35	3.40	0.78
	Pivotal	0.56	0.56	0.55	0.56	0.55	5.88	0.58	0.57	1.62	0.65
200	UCQ	0.63	1.74	0.66	1.00	0.64	229.23	0.76	4.68	23.37	0.75
	Pivotal	0.58	0.82	0.56	0.58	0.57	85.00	0.58	0.58	6.56	0.66

# univ.	Rew.	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20
10	UCQ	0.65	0.86	0.648	11.27	Rew TO	SQL Err.	SQL Err.	Rew TO	3.31	4.62
	Pivotal	0.66	0.69	0.66	11.29	1.02	0.66	0.69	Ans TO	0.56	1.36
200	UCQ	0.60	0.82	0.68	173.51	Rew TO	SQL Err.	SQL Err.	Rew TO	3.30	17.58
	Pivotal	0.67	0.66	0.65	168.45	4.71	0.95	0.68	Ans TO	0.58	5.58

Table 1. Evaluation time over LUBM$^{\exists}_{20}$ (in seconds)

6 Utility Tools for Existential Rules

In this section, we present utility tools dedicated to existential rules, which allow to exchange, decompose and analyze rule bases.

Datalog+ Format and OWL 2 translator We defined a textual format, called *dlgp* (for Datalog+), which extends standard datalog notation. Example 15 shows part of the running example in dlgp format.

In addition to 'pure' existential rules, dlgp allows to encode negative constraints (existential rules with an empty head, interpreted as always false), equality atoms anywhere in the rule bodies and heads (which allows for instance to encode functional dependencies), as well as conjunctive queries and facts. Note that the tools currently implemented in Graal do not process negative constraints and rules with equality in a specific way. For compatiblity with semantic web languages, the use of URIs instead of standard predicates or term names is allowed.

Graal is provided with a dlgp parser and writer. It comes also with a translator of OWL 2, built on the OWL API. This tool processes OWL 2 axioms that can be translated into existential rules and ignores the others.

Example 15 (Rules R_{8a} and R_{8b} in dlgp format).

```
[R8a]criticalManager(X):-isManagerOf(X,Y),hasArea(Y,Z),sensitiveArea(Z).
[R8b]isManagerOf(X,Y),hasArea(Y,Z),sensitiveArea(Z):-criticalManager(X).
```

Decomposition tools As already explained, existential variables in rule heads 'glue' atoms into subsets ('pieces') that have to be processed as a whole. Formally, a *piece* P in a rule head H is a non-empty and inclusion-minimal subset of H such that: for all $A \in P$ and A' in H, if A and A' share an existential variable, then $A' \in P$. A rule head is said to be *single-piece* (resp. *atomic*) if it is composed of a single-piece (resp. a single atom).

A piece of a rule R can be seen as a 'unit' of knowledge brought by an application of R. Indeed, R can be decomposed into an equivalent set of single-piece-head rules with the same body; furthermore, a rule with a single-piece-head cannot be decomposed into an equivalent set of atomic-head rules, except by adding a new predicate.

Example 16 (Running example). Rule R_0 has no existential variable, thus each atom forms a piece. It can be decomposed into: $\{R_{0,1}{=}project(x,y,z,w) \rightarrow hasArea(x,y)$, $R_{0,2} = project(x,y,z,w) \rightarrow hasScManager(x,z)$, $R_{0,3} = project(x,y,z,w) \rightarrow hasAdmManager(x,w)\}$.

By adding a special predicate, one can always decompose an existential rule into atomic-head rules. Hence, without loss of expressivity one could restrict attention to such rules. However, breaking the rule pieces has several drawbacks. First, it leads to a less accurate analysis of dependencies between rules. Second, it leads to less efficient query rewriting (we refer the reader to the experiments reported in [KLMT15]). Finally, it can even make the forward chaining infinite because it prevents from detecting some redundancies in the saturated facts as shows Example 17.

Example 17. Consider the rule $R = p(x) \rightarrow r(x,y) \wedge r(y,y) \wedge p(y)$ and its decomposition into atomic-head rules: $\{ R_1 = p(x) \rightarrow p_R(x,y)$, $R_2 = p_R(x,y) \rightarrow r(x,y)$, $R_3 = p_R(x,y) \rightarrow r(y,y)$, $R_4 = p_R(x,y) \rightarrow p(y) \}$. Let $\mathcal{F} = \{p(a)\}$. The restricted chase with R halts on this instance. The first application of R generates $\mathcal{F}_1 = \{p(a), r(a,y_0), r(y_0,y_0), p(y_0)\}$. The next application generates $\mathcal{F}_2 = \{p(a), r(a,y_0)$, $r(y_0,y_0)$, $p(y_0)$, $r(y_0,y_1)$, $r(y_1,y_1)$, $p(y_1)\}$ that folds into $\mathcal{F}_1$ (with both y_0 and y_1 being mapped to y_0).

Graal provides these two transformations of rules for convenience, however only the decomposition into single-piece heads is exploited in reasoning algorithms, since the other one is always less efficient.

Analysis of a rule set Graal also provides a rule base analyser, which was first developed as Kiabora, available online.[4] We briefly explain why such an analysis may be useful. Since ontological query answering is undecidable for general existential rules, neither forward nor backward chaining mechanisms may halt. Therefore, some 'abstract' properties of rule sets have been defined, in relation with the kind of algorithm that halts on rule sets satisfying these properties. These properties are the following:

[4] `http://www.lirmm.fr/kiabora`

– *FES.* A rule set is a *finite expansion set* when, for any fact base $\mathcal{F}$, $\mathcal{F}^*$ is equivalent to a finite fact base (hence, a forward chaining algorithm able to detect equivalent fact bases halts).
– *FUS.* A rule set is a *finite unification set* when, for any CQ Q, the set of all rewritings that can be obtained by a sequence of direct rewritings from Q has a finite cover (hence, a breadth-first query rewriting algorithm that maintains a minimal cover halts).
– *BTS.* A rule set is a *bounded treewidth set* when, for any fact base $\mathcal{F}$, $\mathcal{F}^*$ has bounded treewidth, even if it may be infinite (see [TBMR12] for an algorithm).

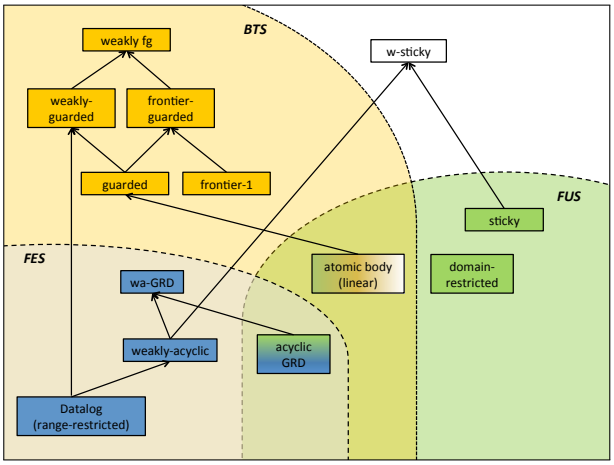

Fig. 1. Decidable classes processed by the analyser

These abstract properties are not recognizable. However, many concrete classes of rules have been exhibited, whose syntactic properties ensure the satisfaction of one or several of the abstract properties. Figure 1 pictures the concrete classes currently recognized by the rule analyser; for an overview of these classes, see, e.g., [Mug11].

Furthermore, the analyser uses the graph of rule dependencies as a tool to improve decidability recognition. Indeed, some global properties on this graph allows one to combine FES, FUS or BTS behaviors to obtain a halting procedure. For instance, if no rule from a subset processed as FES depends on a rule from a subset processed as FUS, one can first saturate the fact base with the FES rules, rewrite the query with the FUS rules, and finally query the partially saturated fact base with the obtained rewritings. For more details on the decidability recognition module, the reader is referred to [LMR13].

7 Conclusion

We presented the main features of Graal, a java toolkit devoted to existential rules and oriented toward ontological query answering. Graal is a modular tool, with minimal dependencies between modules, which allows to embed part of it in another sofware.

It is designed to be easily customized or extended. Its core is a set of java interfaces that can be implemented to plug in other storage sytems, input /output formats, or new algorithms. Future work includes processing negative constraints and equality in rules, providing other exchange formats, such as the Datalog+ fragment of RuleML, and implementing other query answering algorithms as well as approaches allowing to combine them, as initiated in Kiabora rule analyser.

References

BLMS11. J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On Rules with Existential Variables: Walking the Decidability Line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

CDL$^+$07. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

CGL09. A. Calì, G. Gottlob, and T. Lukasiewicz. A General Datalog-Based Framework for Tractable Query Answering over 0ntologies. In *PODS 2009*, pages 77–86, 2009.

CGP10. A. Calì, G. Gottlob, and A. Pieris. Query Answering under Non-guarded Rules in Datalog+/-. In *RR 2010*, pages 1–17, 2010.

CTS11. A. Chortaras, D. Trivela, and G. B. Stamou. Optimized Query Rewriting for OWL 2 QL. In *CADE'11*, pages 192–206, 2011.

FKMP05. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

GOP14. G. Gottlob, G. Orsi, and A. Pieris. Query Rewriting and Optimization for Ontological Databases. *ACM Trans. Database Syst.*, 39(3):25, 2014.

GPH05. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

KLM15. M. König, M. Leclère, and M.-L. Mugnier. Query Rewriting for Existential Rules with Compiled Preorder. In *IJCAI 2015*, 2015.

KLMT15. M. König, M. Leclère, M.-L. Mugnier, and M. Thomazo. Sound, Complete and Minimal UCQ-Rewriting for Existential Rules. *Sem. Web J., to appear*, 2015.

LMR13. M. Leclère, M.-L. Mugnier, and S. Rocher. Kiabora: An Analyzer of Existential Rule Bases. In *RR 2013*, pages 241–246, 2013.

LSTW13. C. Lutz, I. Seylan, D. Toman, and F. Wolter. The Combined Approach to OBDA: Taming Role Hierarchies Using Filters. In *ISWC 2013*, pages 314–330, 2013.

Mug11. M.-L. Mugnier. Ontological query answering with existential rules. In *RR 2011*, pages 2–23, 2011.

PHM09. H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for OWL 2. In *ISWC 2009*, pages 489–504, 2009.

PLC$^+$08. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking Data to Ontologies. *J. Data Semantics*, 10:133–173, 2008.

RC12. M. Rodriguez-Muro and D. Calvanese. High Performance Query Answering over DL-Lite Ontologies. In *KR 2012*, 2012.

RMKZ13. M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Query Rewriting and Optimisation with Database Dependencies in Ontop. In *DL'13*, pages 917–929, 2013.

TBMR12. M. Thomazo, J.-F. Baget, M.-L. Mugnier, and S. Rudolph. A Generic Querying Algorithm for Greedy Sets of Existential Rules. In *KR 2012*, 2012.

TSCS13. D. Trivela, G. Stoilos, A. Chortaras, and G. B. Stamou. Optimising Resolution-Based Rewriting Algorithms for DL Ontologies. In *DL'13*, pages 464–476, 2013.