Exploiting Conjunctive Queries in Description Logic Programs

Thomas Eiter
Thomas Krennwallner
Roman Schindlauer
Institut für Informationssysteme, TU Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,tkren,roman}@kr.tuwien.ac.at

Giovambattista Ianni
Dip. di Matematica, Università della Calabria,
I-87036 Rende (CS), Italy.
ianni@mat.unical.it

Dedicated to Victor Marek
on his 65th birthday

Abstract

We present cq-programs, which enhance nonmonotonic description logics (dl-) programs by conjunctive queries (CQ) and union of conjunctive queries (UCQ) over Description Logics knowledge bases, as well as disjunctive rules. dl-programs had been proposed as a powerful formalism for integrating nonmonotonic logic programming and DL-engines on a clear semantic basis. The new cq-programs have two advantages. First, they offer increased expressivity by allowing general (U)CQs in the body. And second, this combination of rules and ontologies gives rise to strategies for optimizing calls to the DL-reasoner, by exploiting (U)CQ facilities of the DL-reasoner. To this end, we discuss some equivalences which can be exploited for program rewriting and present respective algorithms. Experimental results for a cq-program prototype show that this can lead to significant performance improvements.

1 Introduction

Rule formalisms that combine logic programming with other sources of knowledge, especially terminological knowledge expressed in Description Logics (DLs), have gained increasing interest in the past years. This process was mainly fostered by current efforts in the Semantic Web development of designing a suitable rules layer on top of the existing ontology layer. Such couplings between DLs (in the form of ontologies) and logic programming appear in different flavors, which roughly can be categorized in (i) systems with strict semantic integration and (ii) systems with strict semantic separation, which amounts to coupling heterogeneous systems (Rosati 2006a; Eiter *et al.* 2006; Antoniou *et al.* 2005; Pan *et al.* 2004). In this paper, we will concentrate on the latter, considering ontologies as an external source of information with semantics treated independently from the logic program. One representative of this category was presented in (Eiter *et al.* 2004; 2006), extending the answer-set semantics of logic programs towards so-called *dl-programs*. dl-programs consist of a DL part L and a rule part P, and allow queries from P to L. These queries are facilitated by a special type of atoms, which also permit to hypothetically enlarge the assertional part of L with facts imported from the logic program P, thus allowing for a bidirectional flow of information.

The types of queries expressible by dl-atoms in (Eiter *et al.* 2004; 2006) are concept and role membership queries, as well as subsumption queries. Since the semantics of logic programs is usually defined over a domain of explicit individuals, this approach may fail to derive certain consequences, which are implicitly contained in L. Consider the following simplified version of an example from (Motik, Sattler, & Studer 2005):

$$L = \left\{ \begin{array}{l} hates(Cain, Abel), hates(Romulus, Remus), \\ father(Cain, Adam), father(Abel, Adam), father \sqsubseteq \\ parent, \exists father.\exists father^{-}.\{Remus\}(Romulus) \end{array} \right\}$$

$$P = \left\{ \begin{array}{l} BadChild(X) \leftarrow \mathrm{DL}[parent](X, Z), \\ \quad \mathrm{DL}[parent](Y, Z), \mathrm{DL}[hates](X, Y) \end{array} \right\}$$

Apart from the explicit facts, L states that each $father$ is also a $parent$ and that Romulus and Remus have a common father. The single rule in P specifies that an individual hating a sibling is a $BadChild$. From this dl-program, $BadChild(Cain)$ can be concluded, but not $BadChild(Romulus)$, though it is implicitly stated that $Romulus$ and $Remus$ have the same father.

The reason is that, in a dl-program, variables must be instantiated over its Herbrand base (containing the individuals in L and P), and thus unnamed individuals, like the father of Romulus and Remus, are not considered. In essence, this means that dl-atoms only allow for building conjunctive queries that are *DL-safe* in the spirit of (Motik, Sattler, & Studer 2005), which ensures that all variables in the query can be instantiated to named individuals. While this was mainly motivated by retaining decidability of the formalisms, unsafe conjunctive queries are admissible under certain conditions (Rosati 2006a). In this vein, we extend dl-programs by permitting conjunctive queries or unions thereof, to L as first-class citizens in the language. In our example, we may use

$$P' = \{BadChild(X) \leftarrow \mathrm{DL}\left[\begin{array}{c} parent(X, Z), \\ parent(Y, Z), \\ hates(X, Y) \end{array}\right](X, Y)\},$$

where the body of the rule is a CQ $\{parent(X, Z), parent(Y, Z), hates(X, Y)\}$ to L with distinguished variables X and Y. Then we shall obtain the desired result, that $BadChild(Romulus)$ is concluded.

The extension of dl-programs to cq-programs, introduced in this paper, has some attractive features.

• First and foremost, the expressiveness of the formalism is increased significantly, since existentially quantified and therefore unnamed individuals can be respected in query answering through the device of (u)cq-atoms.

• In addition, cq-programs have the nice feature that the integration of rules and the ontology is decidable whenever answering (U)CQs over the ontology (possibly extended with assertions) is decidable. In particular, recent results on the decidability of answering (U)CQs for expressive DLs can be exploited in this direction (Ortiz de la Fuente, Calvanese, & Eiter 2006b; 2006a; Glimm *et al.* 2007). Furthermore, it also allows to express, via conjunction of cq-atoms and negated cq-atoms in rule bodies, certain decidable conjunctive queries with negations; note that negation leads quickly to undecidability (Rosati 2007).

• The availability of CQs opens the possibility to express joins in different, equivalent ways and therefore to the design of a module using automatic rewriting techniques. Such module, starting from a given program (L, P), might produce an equivalent, yet more efficient, program (L, P').

Example 1.1 Both

$$r : BadParent(Y) \leftarrow \mathrm{DL}[parent](X, Y), \mathrm{DL}[hates](Y, X)$$

and

$$r' : BadParent(Y) \leftarrow \mathrm{DL}[parent(X, Y), hates(Y, X)](X, Y)$$

equivalently single out (not necessarily all) bad parents. Here, in r the join between $parent$ and $hates$ is performed in the logic program, while in r' it is performed on the DL-side.

DL-reasoners including RACER, KAON2, and Pellet increasingly support answering CQ. This can be exploited to push joins of multiple atoms from the rule part to the DL-reasoner, or vice versa. Multiple calls to the DL-reasoner are an inherent bottleneck in evaluating cq-programs. Reducing the number of calls can significantly improve performance.

Motivated by the last aspect, we then focus on the following contributions.

• We present a suite of equivalence-preserving transformation rules, by which rule bodies and rules involving (u)cq-atoms can be rewritten. Based on these rules, we then describe algorithms which transform a given cq-program P into an equivalent, optimized cq-program P'.

• We report an experimental evaluation of such rewriting techniques, based on a prototype implementation of cq-programs using dlvhex (Eiter *et al.* 2005) and RACER. It shows the effectiveness of the techniques, and that significant performance increases can be gained. The experimental results are interesting in their own right, since they shed light on combining conjunctive query results from a DL-reasoner.

The experimental prototype for cq-programs is ready for use (see Section 5). To our knowledge, it is currently the most expressive implementation of a system integrating nonmonotonic rules and ontologies.

2 dl-Atoms with Conjunctive Queries

We assume familiarity with Description Logics (DLs) (cf. (Baader *et al.* 2003)), in particular $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$.[1] A DL-KB L is a finite set of axioms in the respective DL. We denote logical consequence of an axiom α from L by $L \models \alpha$.

As in (Eiter *et al.* 2004; 2006), we assume a function-free first-order vocabulary Φ of nonempty finite sets $\mathcal{C}$ and $\mathcal{P}$ of constant resp. predicate symbols, and a set $\mathcal{X}$ of variables. As usual, a *classical literal* (or *literal*), l, is an atom a or a negated atom $\neg a$.

Syntax Informally, a cq-program consists of a DL-KB L and a generalized disjunctive program P, which may involve queries to L. Roughly, such a query may ask whether a specific description logic axiom, a conjunction or a union of conjunctions of DL axioms is entailed by L or not.

A *conjunctive query* (CQ) $q(\vec{X})$ is an expression $\{\vec{X} \mid Q_1(\vec{X}_1), \ldots, Q_n(\vec{X}_n)\}$, where each Q_i is a concept or role expression and each $\vec{X}_i$ is a singleton or pair of variables and individuals, and where $\vec{X} \subseteq \bigcup_{i=1}^{n} \mathrm{vars}(\vec{X}_i)$ are its *distinguished* (or *output*) variables. Intuitively, $q(\vec{X})$ is a conjunction $Q_1(\vec{X}_1) \wedge \cdots \wedge Q_n(\vec{X}_n)$ of concept and role expressions, which is true if all conjuncts are satisfied, and then it is projected on $\vec{X}$. We will omit $\vec{X}$ if it is clear from the context.

A *union of conjunctive queries* (UCQ) $q(\vec{X})$ is a disjunction $\bigvee_{i=1}^{m} q_i(\vec{X})$ of CQs $q_i(\vec{X})$. Intuitively, $q(\vec{X})$ is satisfied, whenever some $q_i(\vec{X})$ is satisfied.

Example 2.1 Regarding our opening example, $cq_1(X, Y) = \{X, Y \mid parent(X, Z), parent(Y, Z), hates(X, Y)\}$ and $cq_2(X, Y) = \{X, Y \mid father(X, Y), father(Y, Z)\}$ are CQs with output X, Y, and $ucq(X, Y) = cq_1(X, Y) \vee cq_2(X, Y)$ is a UCQ.

A dl-atom α is in form $\mathrm{DL}[\lambda; q](\vec{X})$, where $\lambda = S_1 op_1 p_1, \ldots, S_m\ op_m\ p_m$ ($m \geq 0$) is a list of expressions $S_i\ op_i\ p_i$ called *input list*, each S_i is either a concept or a role, $op_i \in \{\uplus, \cup, \cap\}$, p_i is a predicate symbol matching the arity of S_i, and q is a (U)CQ with output variables $\vec{X}$ (in this case, α is called a *(u)cq-atom*), or $q(\vec{X})$ is a dl-query. Each p_i is an *input predicate symbol*; intuitively, $op_i = \uplus$ increases S_i by the extension of p_i, while $op_i = \cup$ increases $\neg S_i$; $op_i = \cap$ constrains S_i to p_i.

Example 2.2 The cq-atom

$$\mathrm{DL}[parent \uplus p; parent(X, Y), parent(Y, Z)](X, Z)$$

with output X, Z extends L by adding the extension of p to the role $parent$, and then joins $parent$ with itself.

A *cq-rule* r is of the form $a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$, where every a_i is a literal and every b_j is either a literal or a dl-atom. We define $H(r) = \{a_1, \ldots, a_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where

[1] We focus on these DLs because they underly OWL-Lite and OWL-DL. Conceptually, cq-programs can be defined for other DLs as well.

$B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*. A *cq-program* $KB = (L, P)$ consists of a DL-KB L and a finite set of cq-rules P.

Example 2.3 Let $KB = (L, P)$, where L is the well-known wine ontology[2] and P is as follows:

$$
\begin{aligned}
visit(L) \vee \neg visit(L) \leftarrow\ & \mathrm{DL}[WhiteWine](W), \\
& \mathrm{DL}[RedWine](R), \\
& \mathrm{DL}[locatedIn](W, L), \\
& \mathrm{DL}[locatedIn](R, L), \\
& not\ \mathrm{DL}\big[locatedIn(L, L')\big](L). \\
\leftarrow\ & visit(X), visit(Y), X \neq Y. \\
some\_visit \leftarrow\ & visit(X). \\
\leftarrow\ & not\ some\_visit. \\
delicate\_region(W) \leftarrow\ & visit(L), delicate(W), \\
& \mathrm{DL}[locatedIn](W, L). \\
delicate(W) \leftarrow\ & \mathrm{DL}[hasFlavor](W, \textit{wine:Delicate}).
\end{aligned}
$$

Informally, the first rule selects a maximal region in which both red and white wine grow, and the next three rules make sure that exactly one such region is picked, by enforcing that no more than two regions are chosen (second rule) and that at least one is chosen (third and fourth) rule. The last two rules single out all the sub-regions of the selected region producing some delicate wine, i.e., if a wine has a delicate flavor which is specified by individual *wine:Delicate*.

Note that the program P exclusively uses instance retrieval queries—with one exception in the first rule: the weakly negated dl-atom is a conjunctive query with only one query atom, since we have to remove the non-distinguished variable L' from the output to keep the rule safe. The program will be used throughout the paper for demonstrating our rewriting methods.

Semantics For any CQ $q(\vec{X}) = \{\ \vec{X} \mid Q_1(\vec{X_1}), \ldots, Q_n(\vec{X_n})\}$, let $\phi_q(\vec{X}) = \exists \vec{Y} \bigwedge_{i=1}^{n} Q_i(\vec{X_i})$, where $\vec{Y}$ are the variables not in $\vec{X}$, and for any UCQ $q(\vec{X}) = \bigvee_{i=1}^{m} q_i(\vec{X})$, let $\phi_q(\vec{X}) = \bigvee_{i=1}^{m} \phi_{q_i}(\vec{X})$. Then, for any (U)CQ $q(\vec{X})$, the set of *answers of* $q(\vec{X})$ *on* L is the set of tuples $ans(q(\vec{X}), L) = \{\vec{c} \in \mathcal{C}^{|\vec{X}|} \mid L \models \phi_q(\vec{c})\}$.

Let $KB = (L, P)$ be a cq-program. The *Herbrand base* of P, denoted HB_P, is the set of all ground literals with a standard predicate symbol that occurs in P and constant symbols in $\mathcal{C}$. An *interpretation* I relative to P is a consistent subset of HB_P. We say I is a *model* of $l \in HB_P$ under L, or I *satisfies* l under L, denoted $I \models_L l$, iff $l \in I$.

A ground dl-atom $a = \mathrm{DL}[\lambda; Q](\vec{c})$ is satisfied w.r.t. I, denoted $I \models_L a$, if $L \cup \lambda(I) \models Q(\vec{c})$, where $\lambda(I) = \bigcup_{i=1}^{m} A_i$ and

- $A_i(I) = \{S_i(\vec{e}) \mid p_i(\vec{e}) \in I\}$, for $op_i = \uplus$;
- $A_i(I) = \{\neg S_i(\vec{e}) \mid p_i(\vec{e}) \in I\}$, for $op_i = \uplus$;
- $A_i(I) = \{\neg S_i(\vec{e}) \mid p_i(\vec{e}) \in I$ does not hold$\}$, for $op_i = \cap$.

Now, given a ground instance $a(\vec{c})$ of a (u)cq-atom $a(\vec{X}) = \mathrm{DL}[\lambda; q](\vec{X})$ (i.e., all variables in $q(\vec{X})$ are replaced by constants), I satisfies $a(\vec{c})$, denoted $I \models_L a(\vec{c})$, if $\vec{c} \in ans(q(\vec{X}), L \cup \lambda(I))$.

Let r be a ground cq-rule. We define (i) $I \models_L H(r)$ iff there is some $a \in H(r)$ such that $I \models_L a$, (ii) $I \models_L B(r)$ iff $I \models_L a$ for all $a \in B^+(r)$ and $I \not\models_L a$ for all $a \in B^-(r)$, and (iii) $I \models_L r$ iff $I \models_L H(r)$ whenever $I \models_L B(r)$. We say that I is a *model* of a cq-program $KB = (L, P)$, or I *satisfies* KB, denoted $I \models KB$, iff $I \models_L r$ for all $r \in ground(P)$. We say KB is *satisfiable* (resp., *unsatisfiable*) iff it has some (resp., no) model. The (strong) answer sets of KB, which amount to particular models of KB, are then defined as usual (cf. (Eiter *et al.* 2004; 2006)).

Example 2.4 The region program KB from Ex. 2.3 has the following three answer sets (only the positive facts of predicates $delicate\_region$ and $visit$ are listed, which are abbreviated by dr resp. v): M_1 = {*dr*(*MountadamRiesling*), *v*(*AustralianRegion*), ...}, M_2 = {*dr*(*LaneTannerPinotNoir*), *dr*(*WhitehallLanePrimavera*), *v*(*USRegion*), ...}, and M_3 = {*dr*(*StonleighSauvignonBlanc*), *v*(*NewZealandRegion*), ...}.

The semantics for cq-programs without $\cap$ can be equivalently defined in terms of HEX-programs (see (Eiter *et al.* 2005)).[3] Furthermore, many of the properties of dl-programs are naturally inherited to cq-programs, like the existence of unique answer set in absence of $\cap$ and not, or if not is used in a stratified way.

The example in the introduction shows that cq-programs are more expressive than dl-programs in (Eiter *et al.* 2004; 2006). Furthermore, answer set existence for KB and reasoning from the answer sets of KB is decidable if (U)CQ-answering on L is decidable, which is feasible for quite expressive DLs including $\mathcal{SHIQ}$ and fragments of $\mathcal{SHOIN}$, cf. (Ortiz de la Fuente, Calvanese, & Eiter 2006b; 2006a; Glimm *et al.* 2007). Rosati's well-known $\mathcal{DL}{+}log$ formalism (Rosati 2006b; 2006a), and the more expressive hybrid MKNF knowledge bases (Motik *et al.* 2006; Motik & Rosati 2007) are closest in spirit to dl- and cq-programs, since they support nonmonotonic negation and use constructions from nonmonotonic logics. However, their expressiveness seems to be different from dl- and cq-programs. It is reported in (Motik *et al.* 2006) that dl-programs (and hence also cq-programs) can not be captured using MKNF rules. In turn, the semantics of $\mathcal{DL}{+}log$-programs inherently involves deciding containment of CQs in UCQs, which seems to be inexpressible in cq-programs.

In the remainder of this paper, however, we focus on equivalence preserving rewritings of (u)cq-atoms, which can be exploited for program optimization.

[2] `http://www.w3.org/TR/owl-guide/wine.rdf`

[3] For space reasons, we omit here the technical reasons of this partial equivalence.

3 Rewriting Rules for (u)cq-Atoms

As shown in Ex. 1.1, in cq-programs we might have different possibilities for defining the same query. Indeed, the rules r and r' there are equivalent over any knowledge base L. However, the evaluation of r' might be implemented by performing the join between $parent$ and $hates$ on the DL side in a single call to a DL-reasoner, while r can be evaluated performing the join on the logic program side, over the results of two calls to the DL-reasoner. In general, making more calls is more costly, and thus r' may be preferable from a computational point of view. Moreover, the size of the result transferred by the single call in this rule r' is smaller than the results of the two calls.

Towards exploiting such rewriting, we present some transformation rules for replacing a rule or a set of rules in a cq-program with another rule or set of rules, while preserving the semantics of the program (see Table 1). By means of (repeated) rule application, we can transform the program into another, equivalent program, which we consider in the next section. Indeed, a rewriting module is conceivable, which rewrites a given cq-program (L, P) into a refined, equivalent cq-program (L, P'), which can be evaluated more efficiently. Note that as for rule application, any ordinary dl-atom $\mathrm{DL}[\lambda; Q](\vec{t})$, where $\vec{t}$ is a non-empty list of terms, is equivalent to the cq-atom $\mathrm{DL}[\lambda; Q(\vec{t})](\vec{X})$, where $\vec{X} = \mathrm{vars}(\vec{t})$.

In the rewriting rules, the input lists λ_1 and λ_2 are assumed to be semantically equivalent (denoted $\lambda_1 \doteq \lambda_2$), that is, $\lambda_1(I) = \lambda_2(I)$, for every Herbrand interpretation I. This means that λ_1 and λ_2 modify the same concepts and roles with the same predicates in the same way; this can be easily recognized (in fact, in linear time). More liberal but more expensive notions of equivalence, taking L and/or P into account, might be considered.

Query Pushing (A) By this rule, cq-atoms $\mathrm{DL}[\lambda_1; cq_1](\vec{Y_1})$ and $\mathrm{DL}[\lambda_2; cq_2](\vec{Y_2})$ in the body of a rule (A1) can be merged. In rule (A2), cq_1' and cq_2' are constructed by renaming variables in cq_1 and cq_2 as follows. Let $\vec{Z_1}$ and $\vec{Z_2}$ be the non-distinguished (i.e., existential) variables of cq_1 and cq_2, respectively. Rename each $X \in \vec{Z_1}$ occurring in cq_2 and each $X \in \vec{Z_2}$ occurring in cq_1 to a fresh variable. Then $cq_1' \cup cq_2'$ is the CQ given by all the atoms in both CQs.

Example 3.1 The rule

$$a \leftarrow \mathrm{DL}[R_1(X,Y), R_2(Y,Z)](X), \mathrm{DL}[R_3(X,Y)](X,Y)$$

is equivalent to the rule

$$a \leftarrow \mathrm{DL}[R_1(X,Y'), R_2(Y',Z), R_3(X,Y)](X,Y).$$

Query Pushing can be similarly done when cq_1 and cq_2 are UCQs; here, we simply distribute the subqueries and form a single UCQ.

Variable Elimination (B) Suppose an output variable X of a cq-atom in a rule r of form (B1a) or (B1b) occurs also in an atom $X = t$. Assume that t is different from X and that, in case of form (B1a) the underlying DL-KB is under Unique Name Assumption (UNA) whenever t is an output variable. Then, we can eliminate X from r as follows. Standardize the non-output variables of cq-atoms apart from the other variables in r, and replace uniformly X with t in cq, B, and H; let $cq_{X/t}$, $B_{X/t}$, and $H_{X/t}$ denote the respective results. Remove X from the output $\vec{Y}$ and, if t is a variable Z, add Z to them; the resulting rule r', in (B2) is then

QUERY PUSHING

$$r:\ H \leftarrow \mathrm{DL}[\lambda_1; cq_1](\vec{Y_1}), \mathrm{DL}[\lambda_2; cq_2](\vec{Y_2}), B. \quad \text{(A1)}$$

$$r':\ H \leftarrow \mathrm{DL}\big[\lambda_1; cq_1' \cup cq_2'\big](\vec{Y_1} \cup \vec{Y_2}), B. \quad \text{(A2)}$$

where $\lambda_1 \doteq \lambda_2$.

VARIABLE ELIMINATION

$$r_1:\ H \leftarrow \mathrm{DL}[\lambda_1; cq \cup \{X = t\}](\vec{Y}), B. \quad \text{(B1a)}$$

$$r_2:\ H \leftarrow \mathrm{DL}[\lambda_1; cq](\vec{Y}), X = t, B. \quad \text{(B1b)}$$

$$r': H_{X/t} \leftarrow \mathrm{DL}\big[\lambda_2; cq_{X/t}\big](\vec{Y} \setminus \{X\} \cup \omega(t)), B_{X/t}. \quad \text{(B2)}$$

where $\lambda_1 \doteq \lambda_2$, $X \in \vec{Y}$, $\cdot_{X/t}$ denotes replacement of variable X by t, and $\omega(t) = \{Z\}$ if t is a variable Z and $\omega(t) = \emptyset$ otherwise.

INEQUALITY PUSHING

$$r:\ H \leftarrow \mathrm{DL}[\lambda_1; cq](\vec{Y}), X \neq t, B. \quad \text{(C1)}$$

$$r':\ H \leftarrow \mathrm{DL}[\lambda_2; cq \cup \{X \neq t\}](\vec{Y}), B. \quad \text{(C2)}$$

where $\lambda_1 \doteq \lambda_2$ and $X \in \vec{Y}$. If t is a variable, then also $t \in \vec{Y}$.

FACT PUSHING

$$\bar{P} = \left\{ \begin{array}{l} f(\vec{c_1}), f(\vec{c_2}), \ldots, f(\vec{c_l}), \\ \quad H \leftarrow \mathrm{DL}[\lambda_1; ucq](\vec{Y}), f(\vec{Y'}), B. \end{array} \right\} \quad \text{(D1)}$$

$$\bar{P'} = \left\{ \begin{array}{l} f(\vec{c_1}), f(\vec{c_2}), \ldots, f(\vec{c_l}), \\ \quad H \leftarrow \mathrm{DL}[\lambda_2; ucq'](\vec{Y}), B. \end{array} \right\} \quad \text{(D2)}$$

where $\lambda_1 \doteq \lambda_2$, $\vec{c_j}$ are ground, $\vec{Y'} \subseteq \vec{Y}$, $ucq = \bigvee_{i=1}^{r} cq_i$, and $ucq' = \bigvee_{i=1}^{r} \Big(\bigvee_{j=1}^{l} cq_i \cup \{\vec{Y'} = \vec{c_j}\} \Big)$.

Let H, H', H_i be heads, B, B', B_i be bodies, and r be a rule of form $H \leftarrow a(\vec{Y}), B$.

UNFOLDING

$$\bar{P} = \{r\} \cup \{H' \vee a(\vec{Y'}) \leftarrow B'.\} \quad \text{(E1)}$$

$$\bar{P'} = \bar{P} \cup \{H'\theta \vee H\theta \leftarrow B'\theta, B\theta.\} \quad \text{(E2)}$$

where θ is the mgu of $a(\vec{Y})$ and $a(\vec{Y'})$ (thus $a(\vec{Y}\theta) = a(\vec{Y'}\theta)$).

COMPLETE UNFOLDING

$$P = Q \cup \{r\} \cup \{\ r_i : H_i \vee a(\vec{Y_i}) \leftarrow B_i.\} \quad \text{(F1)}$$

$$P' = (P \setminus \{r\}) \cup \{\ r_i' : H_i\theta_i \vee H\theta_i \leftarrow B_i\theta_i, B\theta_i.\} \quad \text{(F2)}$$

where $1 \leq i \leq l$, Q has no rules of form r, r_i, no $a(\vec{Z}) \in H_i$ is unifiable with $a(\vec{Y})$, and θ_i is the mgu of $a(\vec{Y})$ and $a(\vec{Y_i})$ (thus $a(\vec{Y}\theta_i) = a(\vec{Y_i}\theta_i)$).

Table 1: Equivalences ($H = a_1 \vee \cdots \vee a_k$; $B = b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$)

equivalent to the rule r_1 in (B1a) or to the rule r_2 in (B1b). By repeated application of this rule, we may eliminate multiple output variables of a cq-atom. Note that variables X in equalities $X = t$ not occurring in any output list can always be eliminated by simple replacement.

Example 3.2 The rules

$$r : a(X, Y) \leftarrow \mathrm{DL}[R(X, Z), C(Y), X = Y](X, Y), b(Y)$$

and

$$r' : a(Y, Y) \leftarrow \mathrm{DL}[R(Y, Z), C(Y)](Y), b(Y)$$

have the same outcome on every DL-KB L. Here, r' should be preferred due to the lower arity of its cq-atom. Similarly, the rule

$$a(X, Y) \leftarrow \mathrm{DL}[R(X, Z), C(Y), Y = c](X, Y), b(Y)$$

can be simplified to the rule

$$a(X, c) \leftarrow \mathrm{DL}[R(X, Z), C(c)](X), b(c).$$

Inequality Pushing (C) If the DL-engine is used under the UNA and supports inequalities in the query language, we can easily rewrite rules with inequality ($\neq$) in the body by pushing it to the cq-query. A rule of form (C1) can be replaced by (C2).

Example 3.3 Consider the rule

$$\begin{aligned} big(M) \leftarrow & \mathrm{DL}[Wine](W_1), \mathrm{DL}[Wine](W_2), W_1 \neq W_2, \\ & \mathrm{DL}[hasMaker](W_1, M), \mathrm{DL}[hasMaker](W_2, M). \end{aligned}$$

Here, we want to know all wineries producing at least two different wines. We can rewrite above rule, by Query and Inequality Pushing, to the rule

$$big(M) \leftarrow \mathrm{DL}\left[\begin{array}{c} Wine(W_1), Wine(W_2), \\ W_1 \neq W_2, \\ hasMaker(W_1, M), \\ hasMaker(W_2, M) \end{array}\right](M, W_1, W_2).$$

A similar rule works for a ucq-atom $\mathrm{DL}[\lambda; ucq](\vec{Y})$ in place of $\mathrm{DL}[\lambda; cq](\vec{Y})$. In that case, we have to add $\{X \neq t\}$ to each cq_i in $ucq = \bigvee_{i=1}^{m} cq_i$.

Fact Pushing (D) Suppose we have a program with "selection predicates," i.e., facts which serve to select a specific property in a rule. We can push such facts into a ucq-atom and remove the selection atom from the rule body.

Example 3.4 Consider the program P, where we only want to know the children of joe and $jill$:

$$P = \left\{ \begin{array}{l} f(joe).\ f(jill). \\ fchild(Y) \leftarrow \mathrm{DL}[isFatherOf](X, Y), f(X). \end{array} \right\}$$

We may rewrite the program to a more compact one with the help of ucq-atoms:

$$\begin{array}{l} f(joe).\ f(jill). \\ fchild(Y) \leftarrow \mathrm{DL}\left[\begin{array}{l} \left\{\begin{array}{c} isFatherOf(X, Y), \\ X = joe \end{array}\right\} \bigvee \\ \left\{\begin{array}{c} isFatherOf(X, Y), \\ X = jill \end{array}\right\} \end{array}\right](X, Y). \end{array}$$

Such a rewriting makes sense in situations were $isFatherOf$ has many values and thus would lead to query, while uselessly, for all known father-child relationships.

The program $\bar{P}$ in (D1) can be rewritten to $\bar{P}'$ in (D2). In general, a cq-program P such that $\bar{P} \subseteq P$ and f does not occur in heads of rules in $P \setminus \bar{P}$ can be rewritten to $(P \setminus \bar{P}) \cup \bar{P}'$.

Unfolding (E) and Complete Unfolding (F) Unfolding rules is a standard method for partial evaluation of ordinary disjunctive logic programs under answer set semantics, cf. (Sakama & Seki 1997). It can be also applied in the context of cq-programs, with no special adaptation. After folding rules with (u)cq-atoms in their body into other rules, subsequent Query Pushing might be applied. In this way, inference propagation can be shortcut.

The following results state that the above rewritings preserve equivalence. Let $P \equiv_L Q$ denote that (L, P) and (L, Q) have the same answer sets.

Theorem 3.5 *Let r and r' be rules of form $(\Theta 1)$ and $(\Theta 2)$, respectively, $\Theta \in \{A, B, C\}$. Let (L, P) be a cq-program with $r \in P$. Then, $P \equiv_L (P \setminus \{r\}) \cup \{r'\}$.*

Theorem 3.6 *Let $\bar{P}$ and $\bar{P}'$ be rule sets of form $(\Theta 1)$ and $(\Theta 2)$, respectively, $\Theta \in \{D, E\}$. Let (L, P) be a cq-program such that $\bar{P} \subseteq P$. Then, $\bar{P} \equiv_L \bar{P}'$ and $P \equiv_L (P \setminus \bar{P}) \cup \bar{P}'$.*

Theorem 3.7 *Let P and P' be rule sets of form* (F1) *and* (F2)*. Then, $P \equiv_L P'$.*

4 Rewriting Algorithms

Based on the results above, we describe algorithms which combine them into a single module for optimizing cq-programs. The optimization process takes several steps. In each step, a special rewriting algorithm works on the result handed over by the preceding step. Note that, in general, some of the rewriting rules might eliminate some predicate name from a given program. This might not be desired if such predicate names play the role of output predicates. Indeed, it is usual that a program P contains auxiliary rules conceived for importing knowledge from an ontology, or to compute intermediate results, while important information, from the user point of view, is carried by output predicates. We introduce thus a set F of *filter* predicates which are explicitly preserved from possible elimination.

The first step performs unfolding, taking in account the content of F. That is, only literals with a predicate from F are kept.

Algorithm 1 uses the function $factpush(P)$ for Fact Pushing. This function tries to turn a program P into a more efficient one by merging rules according to the equivalences in Section 3. The algorithm also combines filtering and unfolding using $unfold(a, r, r')$, which takes two rules r and r' and returns the unfolding of r' with r w.r.t. a literal a. Note that $do\_unfold(a, r, r', P)$ is a generic function for deciding whether the unfolding of a rule r in r' w.r.t. a given program P and a literal a can be done (or is worth being done); this may be carried out, e.g., using a cost model (as we will see later in this Section). do_unfold may also use, e.g., an internal counter for the numbers of iterations or rule unfoldings, and return false if a threshold is exceeded. Also complete

Algorithm 1: $merge(P, F)$: Merge cq-rules in program P w.r.t. F

Input: Program P, Filter $F = \{p_1, \ldots, p_n\}$
Result: Unfolded program P

repeat
  $P^l = P = factpush(P)$
  $C = \{a, a' \mid \exists r, r' \in P : a' \in H(r'), a \in B^+(r),$ and a' unifiable with $a\}$
  if $C \neq \emptyset$ **then**
    choose $a \in C$
    $P' = \emptyset$
    $R_H = \{r \in P \mid a$ unifies with $a' \in H(r)\}$
    $R_B = \{r \in P \mid a$ unifies with $a' \in B^+(r)\}$
    stop_unfold = true
    forall $r_B \in R_B$ **do**
      forall $r_H \in R_H$ **do**
        if $do\_unfold(a, r_H, r_B, P)$ **then**
          stop_unfold = false
          add r_H and $unfold(a, r_H, r_B)$ to P'
          if $|\{b \in H(r_H)$ *such that* b *unifies with* $a\}| > 1$ **then** add r_B to P'
        else
          add r_H and r_B to P'
        end
      end
    end
    $P = P' \cup (P \setminus (R_B \cup R_H))$
  end
until $P^l = P$ *or stop_unfold is true*
return $filter(P, F)$

unfolding cannot take place if more than one atom in the head of r' can unify with a. The function $filter(P, F)$ eliminates rules which have no influence on the filtered output. Such rules are those of form $H \leftarrow B$ where H is nonempty and has no predicate from F and no literal a unifiable either (i) with some literal in the body of a rule from P, or (ii) with some literal in a disjunctive rule head in P, or (iii) with the opposite of some literal in a rule head in P.

Theorem 4.1 *For a cq-program* (L, P) *and filter* F, $P \equiv_L merge(P, F)$ *w.r.t.* F.

After the unfolding process, we can use Algorithm 2 for optimizing all the different kinds of queries in P. Here, $push(a_1, a_2)$ takes any combination of two dl-, cq-, and ucq-atoms and generates an optimized (u)cq-atom. Similar to do_unfold in Algorithm 1, $do\_push(a_1, a_2)$ is a generic function to decide whether the Query Pushing should take place, i.e., it checks compatibility of the input lists of the atoms and decides whether pushing of a_1 and a_2 should be done.

The last part in this algorithm eliminates variables in the output of dl-, cq-, and ucq-atoms according to Variable Elimination.

Theorem 4.2 *For every cq-program* (L, P), $P \equiv_L RuleOptimizer(P)$.

Algorithm 2: $RuleOptimizer(P)$: Optimize the bodies of all cq-rules in P

foreach $r \in P$ *such that* $B^+(r) \neq \emptyset$ **do**
  choose $b \in B^+(r)$
  $B^+(r) = BodyOptimizer(b, B^+(r) \setminus \{b\}, \emptyset, \emptyset)$
  forall $a = \mathrm{DL}[\lambda; cq](\vec{Y})$ *in* $B^+(r)$ *s.t.* $X = t$ *in* cq *or* $B^+(r)$ **do**
    if $X \notin \vec{Y}$ **then**
      $r = H(r) \leftarrow \mathrm{DL}[\lambda; cq_{X/t}](\vec{Y}), B^+(r) \setminus \{a\}, not\ B^-(r)$
    else
      $r = H(r)_{X/t} \leftarrow \mathrm{DL}[\lambda; cq_{X/t}](\vec{Y} \setminus \{X\} \cup \omega(t)), (B^+(r) \setminus \{a\})_{X/t}, not\ B^-(r)_{X/t}$
    end
  end
end
return P

Algorithm 3: $BodyOptimizer(o, B, C, O)$: Push queries in body B wrt. o

Input: atom o, body B, carry C, and optimized body O
Result: pushed optimized body B

if $B \neq \emptyset$ **then**
  choose $b \in B$
  if $do\_push(o, b)$ **then**
    $o = push(o, b)$
  else
    $C = C \cup \{b\}$
  end
  if $|B| > 1$ **then**
    return $BodyOptimizer(o, B \setminus \{b\}, C, O)$
  else if $|C| \neq \emptyset$ **then**
    choose $c \in C$
    return $BodyOptimizer(c, C \setminus \{c\}, \emptyset, O \cup \{o\})$
  end
end
return $O \cup \{o\}$

Example 4.3 Let us reconsider the region program on the wine ontology in Ex. 2.3. Using the optimization methods for cq-programs we obtain from P an equivalent program P', where the first rule in P is replaced by

$$visit(L) \vee \neg visit(L) \leftarrow \mathrm{DL}\left[\begin{array}{c} WhiteWine(W_1), RedWine(W_2), \\ locatedIn(W_1, L), locatedIn(W_2, L) \end{array}\right](W_1, W_2, L), \\ not\ \mathrm{DL}[locatedIn(L, L')](L),$$

and the second last rule in P is replaced by

$$delicate\_region(W) \leftarrow visit(L), \mathrm{DL}\left[\begin{array}{c} hasFlavor(W, \textsf{wine:Delicate}), \\ locatedIn(W, L) \end{array}\right](W, L).$$

The dl-queries in the first rule were pushed into a single CQ. Furthermore, the rule defining $delicate$ was folded into the

last rule, and subsequently Query Pushing was applied to it.

Cost Based Query Pushing The functions do_unfold and do_push in Alg. 1 and 3 determine whether we can benefit from unfolding or query pushing. Given the input parameters, they should know whether doing the operation leads to a "better" program in terms of evaluation time, size of the program, arity of (u)cq-atoms, data transmission time, etc.

In the database area, cost estimations are based on a cost model, which usually has information about the size of a database and its relations, an estimate of the selectivity of joins and selections, the cost of the data transfer, etc. In our setting, similar knowledge can be used to determine the cost for pushed queries.

Another useful strategy is to exploit knowledge about presence of functional properties in L. A property R is *functional*, if for all individuals x, y_1, y_2 it holds that $R(x, y_1) \wedge R(x, y_2) \rightarrow y_1 = y_2$, i.e., x is a key in R.

Example 4.4 The fact that every person has only one mother may be stated by the functional property $hasMother$, expressed by the axiom $\top \sqsubseteq \leq 1.hasMother$. The following rule retrieves all mothers of men:

$$r : a(Y) \leftarrow \mathrm{DL}[hasMother](X, Y), \mathrm{DL}[Man](X).$$

After application of Query Pushing, we obtain the rule

$$r' : a(Y) \leftarrow \mathrm{DL}[hasMother(X, Y), Man(X)](X, Y).$$

In r we get two answers with size $|hasMother| + |Man|$, while in r' we retrieve at most $|Man|$ tuples. Pushing would be even more effective if the concept was very selective, e.g., if we had $Nobel\_Laureate$ instead of Man.

5 Experimental Results

In this section, we provide experimental results for the rule transformations and the performance gain obtained by applying the various optimization techniques. We have tested the rule transformations using the prototype implementation of the DL-plugin for dlvhex,[4] a logic programming engine featuring higher-order syntax and external atoms (see (Eiter *et al.* 2005)), which uses RACER 1.9 as DL-reasoner (cf. (Haarslev & Möller 2001)). To our knowledge, this is currently the only implemented system for such a coupling of nonmonotonic logic programs and Description Logics.

The tests were done on a P4 3GHz PC with 1GB RAM under Linux 2.6. As an ontology benchmark, we used the testsuite described in (Motik & Sattler 2006). The experiments covered particular query rewritings and version of the region program (Ex. 2.3) with the optimizations applied. We report only part of the results, which are shown in Fig. 1. Missing entries mean memory exhaustion during evaluation.

In most of the tested programs, the performance boost using the aforementioned optimization techniques was substantial. Due to the size of the respective ontologies, in some cases the DL-engines failed to evaluate the original dl-queries, while the optimized programs did terminate with the correct result.

[4] `http://www.kr.tuwien.ac.at/research/dlvhex/`

VICODI program: (Fact Pushing)

$$P_v = \left\{ \begin{array}{l} c(\textit{vicodi:Economics}), c(\textit{vicodi:Social}), \\ v(X) \leftarrow \mathrm{DL}[hasCategory](X, Y), c(Y) \end{array} \right\}$$

SEMINTEC query: (Query Pushing)

$$P_{s_2} = \left\{ \begin{array}{ll} s_2(X, Y, Z) \leftarrow & \mathrm{DL}[Man](X), \\ & \mathrm{DL}[isCreditCard](Y, X), \\ & \mathrm{DL}[Gold](Y), \\ & \mathrm{DL}[livesIn](X, Z), \\ & \mathrm{DL}[Region](Z) \end{array} \right\}$$

SEMINTEC costs: (Query Pushing, Functional Property)

$$P_l = \{l(X, Y) \leftarrow \mathrm{DL}[hasLoan](X, Y), \mathrm{DL}[Finished](Y)\}$$

$hasLoan$ is an inverse functional property and $|hasLoan| = 682(n + 1)$, $|Finished| = 234(n + 1)$, where n is obtained from the ontology instance SEMINTEC_n.

LUBM faculty: (Query Pushing, Inequality Pushing, Variable Elimination)

$$P_f = \left\{ \begin{array}{ll} f(X, Y) \leftarrow & \mathrm{DL}[Faculty](X), D_1 = D_2, \\ & \mathrm{DL}[Faculty](Y), U_1 \neq U_2, \\ & \mathrm{DL}[doctoralDegreeFrom](X, U_1), \\ & \mathrm{DL}[worksFor](X, D_1), \\ & \mathrm{DL}[doctoralDegreeFrom](Y, U_2), \\ & \mathrm{DL}[worksFor](Y, D_2). \end{array} \right\}$$

Table 2: Some test queries

In detail, for the region program, we used the ontologies wine_0 through wine_9. As can be seen from the first graph in Fig. 1, there is a significant speedup, and in case of wine_9 only the optimized program could be evaluated. Most of the computation time was spent by RACER. We note that the result of the join in the first rule had only size linear in the number of top regions L; a higher performance gain may be expected for ontologies with larger joins.

The VICODI test series revealed the power of Fact Pushing (see the second graph in Fig. 1). While the unoptimized VICODI program (Table 2) could be evaluated only with ontologies VICODI_0 and VICODI_1, all ontologies VICODI_0 up to VICODI_4 could be handled with the optimized program.

The SEMINTEC tests dealt with Query Pushing for single rules. The rule in P_{s_2} is from one of the benchmark queries in (Motik & Sattler 2006), while P_l tests the performance increase when pushing a query to a functional property (see Table 2). In both cases, we performed the tests on the ontologies SEMINTEC_0 up to SEMINTEC_4. As shown in Fig. 1 (third graph) the evaluation speedup was significant. We could complete the evaluations of P_{s_2} on all SEMINTEC ontologies only with the optimization. The performance gain for P_l is in line with the constant join selectivity.

In the LUBM test setup, we used the LUBM Data Generator[5] to create the Department ontologies for University 1. We then created 15 ontologies out of this setup, where each ontology LUBM_n has Department 1 up to Department n in the ABox. The test query P_f (Fig. 1, last graph) showed a drastic performance improvement.

[5] `http://swat.cse.lehigh.edu/projects/lubm/`

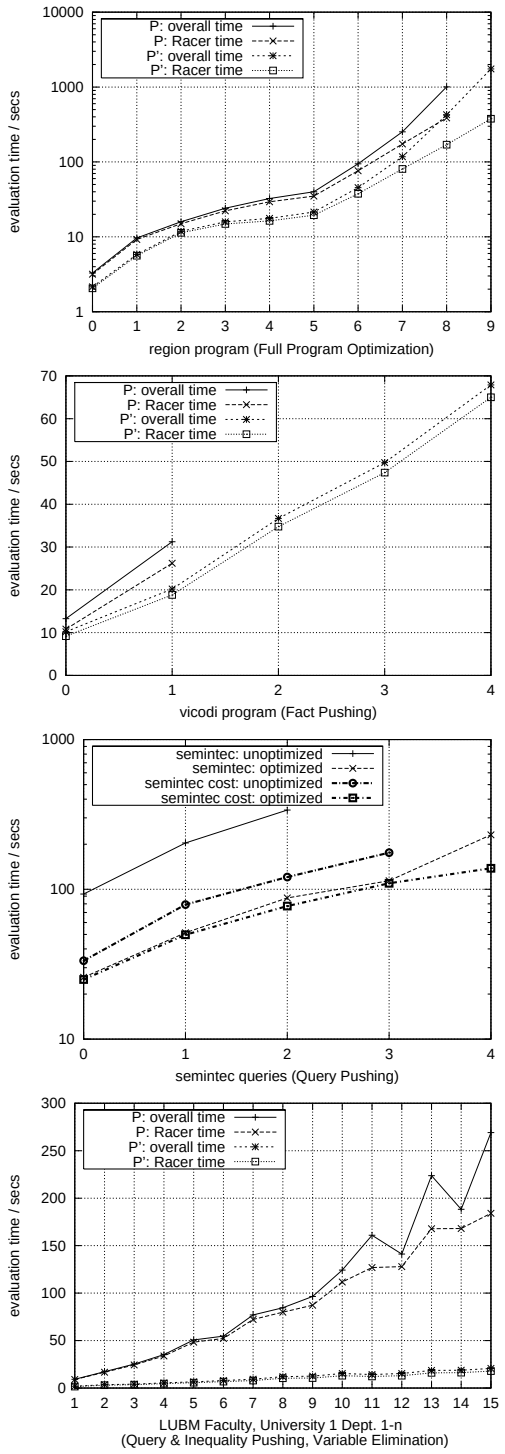

Figure 1: Evaluation time for the examples.

6 Conclusion

We presented cq-programs, which extend dl-programs in (Eiter *et al.* 2004; 2006) with conjunctive queries (CQs) and unions of conjunctive queries (UCQs), to a Description Logic (DL) knowledge base. Such programs have higher expressiveness than dl-programs and retain decidability of reasoning as long as answering CQs resp. UCQs is decidable. As we have explored in this paper, CQs and UCQs can also be exploited for program optimization. By pushing CQs to the highly optimized DL-reasoner, significant speedups can be gained, and in some cases evaluation is only feasible in that way.

The results are promising and suggest to further this path of optimization. To this end, refined strategies implementing the tests do_unfold and do_push are desirable, as well as further rewriting rules. In particular, an elaborated cost model for query answering would be interesting. However, given the continuing improvements on DL-reasoners, such a model had to be revised more frequently.

Future work will be to compare to realizations of cq-programs based on other DL-engines which host CQs, such as Pellet and KAON2, and to enlarge and refine the rewriting techniques.

References

Antoniou, G.; Damásio, C. V.; Grosof, B.; Horrocks, I.; Kifer, M.; Maluszynski, J.; and Patel-Schneider, P. F. 2005. Combining Rules and Ontologies: A survey. Technical Report IST506779/Linköping/I3-D3/D/PU/a1, Linköping University.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P.F.; eds.: The Description Logic Handbook.: Theory, Implementation and Applications. Cambridge University Press (2003)

Eiter, T.; Lukasiewicz, T.; Schindlauer, R.; and Tompits, H. 2004. Combining Answer Set Programming with Description Logics for the Semantic Web. In *Proc. KR-2004*, 141–151.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proc. IJCAI 2005*, 90–97. Morgan Kaufmann.

Eiter, T.; Ianni, G.; Polleres, A.; Schindlauer, R.; and Tompits, H. 2006. Reasoning with Rules and Ontologies. In *Reasoning Web, Summer School 2006*, number 4126 in LNCS, 93–127. Springer.

Glimm, B.; Horrocks, I.; Lutz, C.; and Sattler, U. 2007. Conjunctive Query Answering for the Description Logic $\mathcal{SHIQ}$. In *Proc. IJCAI'07*.

Haarslev, V.; and Möller, R. 2001. RACER System Description. In *Proc. IJCAR-01*, volume 2083 of *LNAI*, 701–705. Springer-Verlag.

Motik, B.; Horrocks, I.; Rosati, R.; and Sattler, U.: Can OWL and Logic Programming Live Together Happily Ever After? In: Proc. ISWC-2006. LNCS 4273, Springer (2006) 501–514

Motik, B.; and Rosati, R.: A Faithful Integration of Description Logics with Logic Programming. In: Proc. IJCAI 2007, AAAI Press/IJCAI (2007) 477–482

Motik, B.; and Sattler, U. 2006. A Comparison of Reasoning Techniques for Querying Large Description Logic

ABoxes. In *Proc. LPAR 2006*, volume 4246 of *LNCS*, 227–241. Springer.

Motik, B.; Sattler, U.; and Studer, R. 2005. Query Answering for OWL-DL with Rules. *Journal of Web Semantics* 3(1):41–60.

Ortiz de la Fuente, M.; Calvanese, D.; and Eiter, T. 2006a. Characterizing Data Complexity for Conjunctive Query Answering in Expressive Description Logics. In *Proc. AAAI '06*. AAAI Press.

Ortiz de la Fuente, M.; Calvanese, D.; and Eiter, T. 2006b. Data Complexity of Answering Unions of Conjunctive Queries in SHIQ. In *Proc. DL2006*, number 189 in CEUR Workshop Proceedings, 62–73.

Pan, J. Z.; Franconi, E.; Tessaris, S.; Stamou, G.; Tzouvaras, V.; Serafini, L.; Horrocks, I. R.; and Glimm, B. 2004. Specification of Coordination of Rule and Ontology Languages. Project Deliverable D2.5.1, KnowledgeWeb NoE.

Rosati, R. 2006a. Integrating Ontologies and Rules: Semantic and Computational Issues. In *Reasoning Web, Summer School 2006*, number 4126 in LNCS, 128–151. Springer.

Rosati, R. 2006b. $\mathcal{DL}$+log: Tight Integration of Description Logics and Disjunctive Datalog. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), 68–78. AAAI Press.

Rosati, R. 2007. The Limits of Querying Ontologies. In *Proc. ICDT 2007*, volume 4353 of *LNCS*, 164–178. Springer.

Sakama, C.; and Seki, H. 1997. Partial Deduction in Disjunctive Logic Programming. *Journal of Logic Programming* 32(3):229–245.