Tighter integration in dl-programs

L. Cruz-Filipe[1,3,4], P. Engrácia[1,3], G. Gaspar[2,4], R. Henriques[2], I. Nunes[2,4], and D. Santos[2]

[1] Escola Superior Náutica Infante D. Henrique, Paço d'Arcos, Portugal
[2] Faculdade de Ciências da Universidade de Lisboa, Portugal
[3] Centro de Matemática e Aplicações Fundamentais, Lisboa, Portugal
[4] LabMag, Lisboa, Portugal

Abstract. We introduce a mechanism called lifting to share predicates between the two components of a dl-program, integrating them in a tighter way. Using lifting, one can reason about the predicates being shared both via the description logic knowledge base and via Datalog-style rules, and the deductions one makes are automatically reflected globally on both components. This is a capability not directly present in dl-programs, since changes to the knowledge base only affect the queries where they occur. We show that lifting has nice theoretical properties, making it suitable for modular design of dl-programs. Furthermore, dl-program processors can easily incorporate lifting as a new operator, and we have extended `dlvhex` to work with dl-programs with lifting.

1 Introduction

For several years now, dl-programs have earned a place in the semantic web community as a convenient language for combining rules and ontologies [5, 6]. The syntax of dl-programs facilitates interaction between ontologies, typically expressed by means of some description logic, and rule-based reasoning, in a logic programming style. The interaction between these two components is achieved by means of *dl-queries*, which function as a bridge: the knowledge base is asked for some conclusion, possibly assuming extra information that it does not already contain. Furthermore, under quite general assumptions, reasoning in dl-programs is decidable and even comes with nice complexity bounds [2, 4, 9, 11].

Although the clean separation between two different worlds – the description logic knowledge base and the rule-oriented program – is usually seen as a positive feature of dl-programs, it has some drawbacks. The flow of information is not symmetric: answers provided by the knowledge base have a permanent effect on the rule-oriented program, but the extra information fed into a given query is local and meant only to extend the knowledge base in the context of that precise query.

However, in some specific scenarios, one might like to globally extend the program's view of the knowledge base. Consider the following situations, where the dl-program is using a pre-existing general ontology as its knowledge base $\mathcal{L}$: (i) additional relations, specific to the context of the dl-program, hold between

the concepts of $\mathcal{L}$; (ii) one wants to apply closed-world reasoning to some concepts or roles of $\mathcal{L}$.

We propose a mechanism, called *lifting*, to obtain a complete two-way integration of the two components of a dl-program. A concept or role from the knowledge base can be lifted to the rule-oriented program, thereby becoming available on both levels of the dl-program; changes made to it on either level will automatically be reflected at the other level, so that lifting effectively identifies a predicate with a concept or role. Thus, lifting achieves tightness (for that predicate) in the sense of [10], but in a controlled way – the user can choose which concepts or roles to lift, unlike in languages where all is shared. The formal concept of lifting abstracts some of the ideas already present in previous work by other authors; however, by studying lifting in itself, we identify a systematic procedure suitable for automation; as such, we have extended the DL-plugin of `dlvhex`, an interpreter for dl-programs, with the capability of accepting dl-programs with lifting, thus making this construction a first-class syntactic operator. We also present some results showing that lifting is modular, and when it is applied correctly there are no unexpected effects on the global program.

The remainder of the paper is structured as follows. Section 2 presents some theoretical notions we will need throughout and discusses some closely related work. Section 3 introduces our lifting construction, motivated by an example, and Section 4 discusses an implementation. The benefits of lifting are summarized in the conclusion, together with some thoughts on future developments.

2 Background and related work

Combining description logics (DLs) with rules has been a leading topic of research for the past 15 years. In this section, we briefly summarize some relevant issues.

Description logic programs or dl-programs [4, 6] are heterogeneous loose coupling combinations of rules and ontologies in the sense that one may distinguish the rule from the ontology part, and rules may contain queries to the ontology part of the program. This connection is achieved through *dl-rules*, which are similar to usual rules in logic programs but make it possible, through *dl-atoms* in rule bodies, to query the knowledge base, possibly modifying it (adding facts) for the duration/purpose of the specific query.

A dl-program is a pair $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, where $\mathcal{L}$ denotes the description logic knowledge base, which we will refer to simply as "knowledge base" from this point onwards, and $\mathcal{P}$ denotes the generalized logic program – a logic program[1] extended with dl-atoms in rules. A dl-atom is of the form

$$DL[S_1\ op_1\ p_1, \ldots, S_m\ op_m\ p_m; R](t)\,,$$

which we often abbreviate to $DL[\chi; R](t)$, where each S_i is either a concept, a role, or a special symbol in $\{=, \neq\}$; $op_i \in \{\uplus, \cup\}$, and the symbols p_i are unary

[1] Throughout this paper, we will assume that the logic programming language is Datalog$^\neg$, described in [9], which in particular contains negation-as-failure.

or binary predicate symbols depending on the corresponding S_i being a concept or a role. $R(t)$ is a dl-query, that is, it is either a concept inclusion axiom F or its negation $\neg F$, or of the form $C(t)$, $\neg C(t)$, $R(t_1, t_2)$, $\neg R(t_1, t_2)$, $= (t_1, t_2)$, $\neq (t_1, t_2)$, where C is a concept, R is a role, t, t_1 and t_2 are terms.

In a dl-atom, $p_1, \ldots, p_m$ are called its *input predicate symbols*. Intuitively, $op_i = \uplus$ (resp., $op_i = \cup$) increases S_i (resp., $\neg S_i$) by the extension of p_i.

A dl-rule r is a normal rule, i.e. $r = a \leftarrow b_1, \ldots, b_k, \mathsf{not}\ b_{k+1}, \ldots, \mathsf{not}\ b_m$, where a is an atom, and $b_1, \ldots, b_m$ are either atoms or dl-atoms.

We illustrate the use of dl-programs by means of the following example from [9].

Example 1. Consider the dl-program $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, where:

$$\mathcal{L}: \quad (\geq 2\ \mathsf{toReview}.\top) \sqsubseteq \mathsf{Overloaded} \qquad (i_1)$$

$$\mathsf{Overloaded} \sqsubseteq \forall \mathsf{supervises}^+.\mathsf{Overloaded} \qquad (i_2)$$

$$\{(\mathsf{a}, \mathsf{b})\} \sqcup \{(\mathsf{b}, \mathsf{c})\} \sqsubseteq \mathsf{supervises} \qquad (i_3)$$

$$\mathcal{P}: \quad \mathsf{good}(X) \leftarrow DL[;\mathsf{supervises}](X, Y), \mathbf{not}\ DL[\mathsf{toReview} \uplus \mathsf{paper}; \mathsf{Overloaded}](Y) \qquad (r_1)$$

$$\mathsf{overloaded}(X) \leftarrow \mathbf{not}\ \mathsf{good}(X) \qquad (r_2)$$

$$\mathsf{paper}(\mathsf{b}, \mathsf{p}_1) \leftarrow \qquad (r_3)$$

$$\mathsf{paper}(\mathsf{b}, \mathsf{p}_2) \leftarrow \qquad (r_4)$$

We briefly recall this program's intended meaning as explained in [9]. Axioms (i_1) and (i_2) indicate that someone who has more than two papers to review is overloaded, and that an overloaded person causes all their supervised persons to be overloaded as well. Axiom (i_3) defines the supervision hierarchy. Rule (r_1) indicates that, if X is supervising Y and Y is not overloaded, then X is a good manager. Rule (r_2) indicates that, if X is not a good manager, then X is overloaded.

The authors of [4, 6] integrated the underlying logics of OWL LITE and OWL DL with normal logic programs, extending two different semantics for ordinary normal programs – answer-set semantics and well-founded semantics. For the purpose of this paper, we will only be concerned with the former, which generalizes answer-set semantics for logic programs.

A dl-program is *positive* if it does not contain negations. A positive dl-program $\langle \mathcal{L}, \mathcal{P} \rangle$ has a minimal model computed by the usual fixpoint construction, where an interpretation I (a subset of the Herbrand base of $\mathcal{P}$ extended with all constants from $\mathcal{L}$) satisfies a ground dl-atom $DL[\chi; Q](\bar{t})$, with $\chi = S_1\ op_1\ p_1, \ldots, S_m\ op_m\ p_m$, if $\mathcal{L}(I; \chi) \models Q(\bar{t})$, where $\mathcal{L}(I; \chi) = \mathcal{L} \cup \bigcup_m^{i=1} A_i(I)$ and, for $1 \leq i \leq m$,

$$A_i(I) = \begin{cases} \{S_i(\bar{e}) \mid p_i(\bar{e}) \in I\}, \text{if } op_i = \uplus \\ \{\neg S_i(\bar{e}) \mid p_i(\bar{e}) \in I\}, \text{if } op_i = \cup \end{cases}$$

Given a dl-program $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, we can obtain a positive dl-program by replacing $\mathcal{P}$ with its *strong dl-transform* $s\mathcal{P}^I_{\mathcal{L}}$ relative to $\mathcal{L}$ and an interpretation I. This is obtained by grounding every rule in $\mathcal{P}$ and then (i) deleting every dl-rule r such that $I \models_{\mathcal{L}} a$ for some default negated a in the body of r, and (ii) deleting from each remaining dl-rule the negative body. The informed reader will recognize this to be a generalization of the Gelfond–Lifschitz reduct. Since $\mathcal{KB}^I = \langle \mathcal{L}, s\mathcal{P}^I_{\mathcal{L}} \rangle$ is a positive dl-program, it has a unique least model $\mathcal{M}_{\mathcal{KB}^I}$. A *strong answer set* of $\mathcal{KB}$ is an interpretation I that coincides with $\mathcal{M}_{\mathcal{KB}^I}$.[2]

One can work with dl-programs by means of `dlvhex` [3], a prototype application for computing models of HEX-programs [7], that is, higher-order logic programs with external atoms. HEX-programs are still based on answer-set semantics, but have a more versatile interface mechanism and introduce higher-order reasoning. The DL-plugin for `dlvhex` simulates the behaviour of dl-programs within this more general framework, providing a rewriter that processes the syntax of dl-atoms, thus allowing the use of `dlvhex` directly as a reasoner for dl-programs. Both `dlvhex` and the DL-plugin are implemented in C++, using RACER [8] as a DL reasoning engine to process OWL DL ontologies.

Giving the dl-program in Example 1 as input to the DL-plugin of `dlvhex`, one obtains a single answer set,

$$\{\mathsf{overloaded(c)}, \mathsf{overloaded(b)}, \mathsf{overloaded(a)}, \mathsf{paper(b, p_2)}, \mathsf{paper(b, p_1)}\},$$

which can easily be seen to be the only answer set of that program.

In [4] a number of encouraging results for dl-programs are presented: (i) data complexity is preserved for dl-programs reasoning under well-founded semantics (complete for P) as long as dl-queries in the program can be evaluated in polynomial time, and (ii) reasoning for dl-programs is first-order rewritable (and thus in LOGSPACE under data complexity) when the evaluation of dl-queries is first-order rewritable and the dl-program is acyclic. These nice results are a consequence of the use of Datalog: complexity of query evaluation on both Datalog and stratified Datalog$^\neg$ programs is data complete for PTIME and program complete for EXPTIME, as shown in [2].

It is interesting to observe that [10] gives a list of other criteria by means of which different approaches to combining description logics and rule-based approaches should be compared. These are the following four aspects: *faithfulness* (the semantics of both components is preserved), *tightness* (both components contribute to the consequences of one another), *flexibility* (the same predicate can be viewed under closed- and open-world interpretation at will) and *decidability*. The author argues that dl-programs are faithful, flexible and decidable, but not tight. The latter is not completely true: the possibility of dynamically increasing extents of concepts or roles in a dl-query means that the logic program effectively sees an extended knowledge base for the purpose of that query, hence some (restricted, local) form of tightness is available. This is actually much more versatile, since one can choose for *each* query how each concept or role should be extended. True tightness, achieved by universally and systematically extending

[2] There is also a *weak* answer-set semantics, which we will not discuss here.

a predicate, can be realized by means of the lifting construction we introduce in this paper.

3 Two-level reasoning using lifting

In order to motivate lifting, we introduce a simple, yet rich, example.

The Bigge Auto Shoppe is a well-established car sales company with branches spread around through the country. Its sales management software includes a description logic knowledge base, managed at headquarters, which includes all the relevant information that is intended to be shared by all the company's dealers, including the kind of fuel each car uses (gas or diesel).

The knowledge base includes two subconcepts of car identifying the type of fuel each one uses, Gas and Diesel, together with the following axioms.

$$\mathsf{Diesel} \sqsubseteq \mathsf{Car} \qquad \mathsf{Gas} \sqsubseteq \mathsf{Car} \qquad \mathsf{Diesel} \sqcap \mathsf{Gas} \sqsubseteq \bot$$

Honest Joe was an independent sales agent who started off dealing in electric cars and used a rule-based program to manage his business. Recently, he decided to join The Bigge Auto Shoppe company, but the company's knowledge base has no information on electric cars (and is not planning to add it soon). In order to integrate his data with the company's, Joe decided to develop a dl-program, since this allowed both systems to communicate while retaining their separate reasoning abilities.

At the same time, since electric cars are cars, much of the reasoning machinery in the central knowledge base is relevant to Joe – all properties that are true of all cars apply to electric cars. In order to proceed, he would like the central knowledge base to reason as if his electric cars were also there.

Lifting is the answer to Honest Joe's problems. All he needs is to introduce a new predicate in his program, which he will call car^+, that will allow him to extend the scope of Car in the central knowledge base. He will also need its negation, which he will call car^- since there is no classical negation in his rule-based system. He then writes the following rules.

$$\mathsf{car}^+(X) \leftarrow DL[;\mathsf{Car}](X) \qquad \mathsf{car}^-(X) \leftarrow DL[;\neg\mathsf{Car}](X) \tag{1}$$

$$\mathsf{car}^+(X) \leftarrow \mathsf{electric}(X) \tag{2}$$

Rules (1) introduce the new predicates and guarantee that car^+ inherits all instances of cars in the knowledge base (and correspondingly for car^-). Rule (2) adds the information that electric cars are cars. In order to obtain the reverse correspondence, namely that all facts about car^+ and car^- are fed back into the knowledge base, a global change to the program is required: in every dl-atom, the program must send all new information to the description logic. So, to the input of each dl-atom one must add $\mathsf{Car} \uplus \mathsf{car}^+$ and $\mathsf{Car} \uplus \mathsf{car}^-$. However, this change must be made not only to the dl-atoms presently in the program, but also to any dl-atom added in the future. This motivates the following definition.

Definition 1. *The* dl-program with lifting $\mathcal{KB}_\Gamma$, *where* $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$ *is a dl-program and* $\Gamma = \{Q_1, \ldots, Q_m\}$ *is a finite set of concepts or roles from* $\mathcal{L}$, *is the dl-program* $\langle \mathcal{L}, \mathcal{P}_\Gamma \rangle$ *where* $\mathcal{P}_\Gamma$ *is obtained from* $\mathcal{P}$ *by:*

- *for every* $Q \in \Gamma$, *adding the rules*[3]

$$q^+(\overline{X}) \leftarrow DL[;Q](\overline{X}) \qquad\qquad q^-(\overline{X}) \leftarrow DL[;\neg Q](\overline{X}) \tag{3}$$

- *replacing every dl-atom* $DL[\chi; R](\bar{t})$ *(including those added in the previous step) with*

$$DL[\chi, Q_1 \uplus q_1^+, Q_1 \uplus q_1^-, \ldots, Q_m \uplus q_m^+, Q_m \uplus q_m^-; R](\bar{t}) \tag{4}$$

  where $\chi \equiv S_1\, op_1\, p_1, \ldots, S_n\, op_n\, p_n$ *corresponds to the original query's input. We will call the query in (4) a* Γ-extended query *and abbreviate it to* $DL_\Gamma[\chi; R](\bar{t})$.

In practice, one can define a dl-program with lifting simply by giving $\mathcal{KB}$ and Γ. In our example, Honest Joe can *lift* the concept Car, obtaining a dl-program with lifting where $\Gamma = \{\mathsf{Car}\}$, and then add rule (2) to $\mathcal{P}$.

In order to establish good programming practice, one should take care to add Q to Γ before enriching $\mathcal{P}$ with any rules involving q^+ or q^-. This is justified by the following result.

Theorem 1. *Let* $\mathcal{KB}_\Gamma = \langle \mathcal{L}, \mathcal{P}_\Gamma \rangle$ *be a dl-program with lifting where no rule in* $\mathcal{P}$ *uses a lifted predicate name in its head. For every dl-atom in* $\mathcal{P}$, *if the knowledge base underlying its query in* $\mathcal{KB}$ *is consistent, then so is the knowledge base underlying it in* $\mathcal{KB}_\Gamma$.

Proof (Sketch.). This proof relies on the observation that, after lifting a concept or role Q, the only "new" ways of proving e.g. $Q(t)$ in $\mathcal{L}$ are those arising from proving $q^+(t)$ in $\mathcal{P}$. In turn, the hypothesis ensures that this can only happen if $Q(t)$ already holds in $\mathcal{L}$. This holds for every lifted concept or role; clearly, non-lifted concepts and roles maintain their semantics. Hence, if some dl-atom has an inconsistent context after lifting, it already did so before lifting.

A more detailed proof of this result can be found in [1]. □

This result states that lifting works as intended. The restriction is essential, since rules using lifted predicates can easily make the knowledge base underlying extended queries inconsistent – just consider the case where $\mathcal{P}$ contains the rule $q^+(X) \leftarrow q^-(X)$.

From the proof of the theorem one also obtains the following result, stating that indeed Q and q^+ have the same semantics.

Corollary 1. *In the conditions of Theorem 1, the sets* $\{\bar{t} \mid Q_i(\bar{t})\}$ *and* $\{\bar{t} \mid q_i^+(\bar{t})\}$ *coincide for every* i, *as well as the sets* $\{\bar{t} \mid \neg Q_i(\bar{t})\}$ *and* $\{\bar{t} \mid q_i^-(\bar{t})\}$.

[3] Note that Q may be a concept or a role, so q^+ and q^- will be unary or binary predicates. The names q^+ and q^- are syntactically derived from Q, with the first letter changed to lowercase to conform with the convention in logic programming.

It is simple to show that these relationships remain valid after adding new rules to the logic program as long as $\{\bar{t} \mid Q_i(\bar{t})\}$ and $\{\bar{t} \mid \neg Q_i(\bar{t})\}$ are interpreted as in an extended query, so Corollary 1 holds in general. In other words, q_i^+ and q_i^- are true counterparts to Q_i and $\neg Q_i$, tightly brought together by the lifting construction.

Lifting allows Honest Joe to relate his predicate $\mathsf{electric}$ with other concepts in the knowledge base. For example, if he wants to state that electric cars do not run on diesel (from which the knowledge base will be able to infer further information), he can extend his dl-program by lifting Diesel and adding the rule

$$\mathsf{diesel}^-(X) \leftarrow \mathsf{electric}(X)\,.$$

Examples of the kind of information the resulting dl-program can infer are: electric cars have windows, since all cars have windows; electric cars do not have glow plugs, since only diesel cars have glow plugs. This example also explains why it is important to add a name for the negation of the concept being lifted.

Also included in The Bigge Auto Shoppe's knowledge base is a role hasInj specifying the manufacturer of the cars' injectors (when they have injectors). All diesel cars come with an injector, while cars running on gas may or may not have an injector. After a few weeks of joining them, Honest Joe decided to start selling custom-made diesel cars produced by his cousin, who is also the local salesperson for Great Injectors Inc. Obviously, all those cars have injectors produced by that company. Honest Joe wants to use his company's knowledge base to reason about these cars too, so he decided to lift hasInj and add the following rules to his program.

$$\mathsf{diesel}^+(X) \leftarrow \mathsf{customCar}(X) \tag{5}$$

$$\mathsf{hasInj}^+(X, \mathsf{greatInj}) \leftarrow \mathsf{customCar}(X) \tag{6}$$

Note that Diesel had already been lifted. From Honest Joe's point of view these concepts or roles will be updated whenever he queries the company's knowledge base. However, these custom cars will not really be present in that knowledge base, since all the changes are actually performed in his part of the dl-program. This is in line with The Bigge Auto Shoppe's policy of only recording information about mainstream cars.

At this stage, Honest Joe has lifted two concepts and one role in sequence, so he is working with a dl-program with lifting where $\Gamma = \{\mathsf{Car}, \mathsf{Diesel}, \mathsf{hasInj}\}$. The example shows how the lifting steps can be made at different points in time, the set Γ keeping track of how dl-atoms are being extended. For example, rules (1) were introduced when $\Gamma = \{\mathsf{Car}\}$; at that time, when processing dl-queries the knowledge base would be given an extended context containing $\mathsf{Car} \uplus \mathsf{car}^+$ and $\mathsf{Car} \uplus \mathsf{car}^-$. Afterwards, when Diesel was added to Γ, the context for the same queries became extended also with $\mathsf{Diesel} \uplus \mathsf{diesel}^+$ and $\mathsf{Diesel} \uplus \mathsf{diesel}^-$. Finally, when custom cars were introduced and hasInj was lifted, this context was again enlarged, this time with $\mathsf{hasInj} \uplus \mathsf{hasInj}^+$ and $\mathsf{hasInj} \uplus \mathsf{hasInj}^-$. In the end, $\mathcal{P}_\Gamma$

contains the following rules.

$$\begin{aligned}
\mathsf{car}^+(X) &\leftarrow \mathsf{electric}(X) & \mathsf{car}^+(X) &\leftarrow DL_\Gamma[;\mathsf{Car}](X)\\
& & \mathsf{car}^-(X) &\leftarrow DL_\Gamma[;\neg\mathsf{Car}](X)\\
\mathsf{diesel}^+(X) &\leftarrow \mathsf{customCar}(X) & \mathsf{diesel}^+(X) &\leftarrow DL_\Gamma[;\mathsf{Diesel}](X)\\
\mathsf{diesel}^-(X) &\leftarrow \mathsf{electric}(X) & \mathsf{diesel}^-(X) &\leftarrow DL_\Gamma[;\neg\mathsf{Diesel}](X)\\
\mathsf{hasInj}^+(X,\mathsf{greatInj}) &\leftarrow \mathsf{customCar}(X) & \mathsf{hasInj}^+(X,Y) &\leftarrow DL_\Gamma[;\mathsf{hasInj}](X,Y)\\
& & \mathsf{hasInj}^-(X,Y) &\leftarrow DL_\Gamma[;\neg\mathsf{hasInj}](X,Y)
\end{aligned}$$

It is interesting to look at the latest additions to the program, namely at rule (6), in two different ways. From an intuitive point of view, this rule is ensuring that the knowledge base keeps its informal "requirement" that it contains all the relevant information about the cars it knows about – and since Honest Joe's custom-made cars have injectors, these facts should be included in the knowledge base.

One can, however, look at this in a different perspective, and see rule (6) as an implementation of an integrity constraint (more specifically, as a *tuple-generating dependency*) on the dl-program, seen (as a whole) as a database: all custom-made cars have injectors from Great Injectors, Inc, so whenever a fact is added to the dl-program about a specific car that is custom-made, one must ensure that the corresponding information about its injector is also added. In this case, since the information can be automatically inferred from what is already known, this constraint can be simply added as a rule.

So far, we have shown how lifting is used in three different situations. The first example shows how, with lifting, one can add instances of concepts or roles defined in the knowledge base via the logic program with a global impact, so that (from a practical perspective) the program's view of the knowledge base is effectively changed. The second exemplifies the addition of negative information to the knowledge base, which may be relevant for some inference rules. The third example shows how the interplay between the two components of the dl-program can be made more symmetric, allowing for rules that extend (again, globally) the knowledge base from information derived in the logic program. The key aspect in all these examples is that we are interested in making *global* changes to the program's view of the knowledge base, and not simply the local changes easily achieved by means of (standard, non-extended) dl-atoms. Other situations where this need arises include adding closed-world reasoning to specific concepts or roles – which can only be done on the logic program' side, since description logics typically use open-world semantics – or, more generally, encoding default rules in dl-programs. In particular, Section 6.1 of [5] discusses closed-world reasoning and introduces a complex mechanism to model the extended closed-world assumption that has some syntactic similarities to lifting. However, the construction is presented therein with that specific goal in mind, and the authors do not study its properties as we have done in this section.

Lifting solves one of the main shortcomings of dl-programs pointed in e.g. [10]: its lack of tightness. Typically, in dl-programs the programmer controls the spe-

cific data being input to each dl-atom, locally extending the knowledge base for the purpose of that specific query. However, until lifting there was no mechanism to make *global* changes to the program's view of the database; using lifting, dl-programs become tight.

The next section discusses how lifting was incorporated in an existing tool for dl-programs.

4 Implementation of lifting

Having defined lifting, it is interesting to integrate it into existing tools for reasoning with dl-programs, allowing users to work in a more structured, robust and modular way: writing the logic program $\mathcal{P}$ and specifying the set Γ of lifted concepts or roles. There are obvious advantages to this approach: not only do the programs become significantly shorter and simpler, making them easier to write and less error-prone, but it also ensures the global properties of the dl-program that have to be maintained. From a programmer's point of view, adding Q to Γ is saying that predicate names q^+ and q^- in $\mathcal{P}$ should have the same meaning as Q and $\neg Q$ in the knowledge base. If lifting were not available, to achieve this identification it would be necessary to keep track of the information that needs to be included in every input context; with lifting, one simply writes the simpler (non-extended) dl-atom, and the bookkeeping is done in the background via Γ. Users still get the option of extending the knowledge base locally (via the usual input predicates in dl-queries) or globally (using lifting).

In order to implement lifting, we added a module to the DL-plugin of `dlvhex` that processes a dl-program with lifting. The set Γ is written as special "lifting clauses" of the form

$$\mathsf{Lfc}(Q, q^+, q^-) \qquad \text{or } \mathsf{Lfr}(Q, q^+, q^-)$$

according to whether Q is a concept or a role. This distinction simplifies the implementation, because it gives the module the necessary information about the arity of q^+ and q^-. This is in line with the syntax of dl-programs in the DL-plugin, which also distinguishes dl-queries on roles and on concepts. Also, the user can choose his own names for q^+ and q^- (which was not the case in Section 3). This feature was added for user-friendliness.

There is another difference between the implementation and Definition 1, which is of a technical nature. Because of the conversion to an HEX-program, `dlvhex` performs a *strong-safety check* on its input, identifying circular dependencies between predicates. It then requires that, whenever a predicate P involved in a circularity appears at the head of a rule, the variables in the arguments of P must appear in the body of the same rule as arguments of a predicate not involved in that circularity. This is circumvented by automatically adding a domain predicate to the generated program without any interference by the user.

The new module does the parsing of these lifting clauses, and translates a dl-program with lifting to a dl-program without lifting, as in Definition 1. The

resulting dl-program is then subject to the usual processing by the DL-plugin, which was further unchanged. In particular, if the input program does not have lifting, then the new module simply passes it along without changes, so the DL-plugin works exactly as without this extension.

We illustrate the extended version of the DL-plugin with the example from Section 3. The following listing shows the logic program that was given as input; the underlying description logic knowledge base is omitted. Note that Honest Joe currently has two types of car for sale that are not in the ontology.

```
Lfc{Car,carP,carM}
Lfc{Diesel,dieselP,dieselM}
Lfr{HasInj,hasInjP,hasInjM}

carP(X) :- electric(X).
dieselM(X) :- electric(X).
dieselP(X) :- customCar(X).
hasInjP(X,greatInjInc) :- customCar(X).

electric(passat).
customCar(auditt).
```

The translated dl-program produced by the module is the following. In the DL-plugin, true negation is represented as `-`, $\uplus$ is written as `+=` and $\cup$ as `-=`. Note that the domain predicate includes facts not only about Honest Joe's cars, but also about those that are present in the description logic knowledge base.

```
carP(X) :- electric(X).
dieselM(X) :- electric(X).
dieselP(X) :- customCar(X).
hasInjP(X,greatInjInc) :- customCar(X).

electric(passat).
customCar(auditt).

domain(greatInjInc).
domain(passat).
domain(auditt).
domain(ecoup).
domain(polo).
domain(jetta).
domain(golf).

carP(X) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
            HasInj += hasInjP, HasInj -= hasInjM; Car](X), domain(X).
carM(X) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
            HasInj += hasInjP, HasInj -= hasInjM; -Car](X), domain(X).
dieselP(X) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
              HasInj += hasInjP, HasInj -= hasInjM; Diesel](X), domain(X).
dieselM(X) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
              HasInj += hasInjP, HasInj -= hasInjM; -Diesel](X), domain(X).
hasInjP(X,Y) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
                HasInj += hasInjP, HasInj -= hasInjM; HasInj](X,Y), domain(X), domain(Y).
hasInjM(X,Y) :- DL[Car += carP, Car -= carM, Diesel += dieselP, Diesel -= dieselM,
                HasInj += hasInjP, HasInj -= hasInjM; -HasInj](X,Y), domain(X), domain(Y).
```

The answer-set computed by the DL-plugin (which is the only answer set of this example) is the following, with the domain predicates omitted.

```
{customCar(auditt), electric(passat), hasInjM(ecoup,polo),
hasInjM(ecoup,jetta), hasInjM(ecoup,golf), hasInjM(polo,ecoup),
hasInjM(polo,jetta), hasInjM(polo,golf), hasInjM(jetta,ecoup),
```

```
hasInjM(jetta,polo), hasInjM(jetta,golf), hasInjM(golf,ecoup),
hasInjM(golf,polo), hasInjM(golf,jetta), hasInjP(auditt,greatInjInc),
dieselM(passat), dieselM(ecoup), dieselM(jetta), dieselP(auditt),
dieselP(polo), dieselP(golf), carP(passat), carP(ecoup), carP(polo),
carP(jetta), carP(golf)}
```

This short example already illustrates the advantages of lifting pointed out earlier: the dl-program effectively used is much more complex than the dl-program with lifting that was written, and without lifting one would have to keep track of the six input context updates that were included in the program's six dl-atoms.

Furthermore, future changes to the program would require adding this same six context updates to every new dl-atom, which one would very easily forget; also, future identifications between $\mathcal{L}$ and $\mathcal{P}$ (i.e. new additions to Γ) would require changing the pre-existing part or the program. Lifting avoids both these issues, making it a very useful programming tool.

5 Conclusions and future work

In this paper we presented a lifting construction that allows effective information sharing between the two components of a dl-program, identifying a concept or role in the description logic knowledge base with a predicate in the logic program, bringing true tightness to dl-programs while at the same time making them more flexible.

Lifting allows global changes to the knowledge base to be implemented from the side of the logic program. This is useful not only in situations where the knowledge base is not accessible for change, but also when the intended changes are not desirable outside the dl-program's specific context. Relevant examples include adding closed-world reasoning to specific concepts or roles, or encoding default rules in dl-programs. Furthermore, lifting presents itself as a simple syntactic mechanism that can make development of dl-programs simpler whenever tight integration is desired.

We have implemented lifting as an add-on to the DL-plugin for `dlvhex`, allowing the input of dl-programs with special lifting clauses. This implementation closely adheres to the theoretical definition, although some practical issues had to be resolved, as discussed in Section 4. This implementation still has some limitations, mostly already existing in the DL-plugin for `dlvhex`; namely, it does not support dealing with namespaces associated with imported ontologies. Another direction of ongoing work is providing support for dealing with namespaces associated with imported ontologies in the DL-plugin for `dlvhex`. We are currently working on empirically evaluating how lifting behaves in more large-scale examples, and how the theoretical, worst-case computational complexity bounds for dl-programs actually translate in the practical performance of dl-programs with lifting.

Acknowledgements

This work was partially supported by Fundação para a Ciência e Tecnologia under contract PEst-OE/EEI/UI0434/2011. Rita Henriques was partially sponsored by a grant from LabMAg. Daniel Santos was sponsored by a grant "Bolsa Universidade de Lisboa / Fundação Amadeu Dias".

References

1. L. Cruz-Filipe, P. Engrácia, G. Gaspar, and I. Nunes. Achieving tightness in dl-programs. Technical report, Faculty of Sciences of the University of Lisbon, 2012. Available at `http://hdl.handle.net/10455/6872`.
2. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
3. The `dlvhex` tool. Available at `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`.
4. T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer. Well-founded semantics for description logic programs in the semantic Web. *ACM Transactions on Computational Logic*, 12(2), 2011. Article Nr 11.
5. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12–13):1495–1539, 2008.
6. T. Eiter, G. Ianni, A. Polleres, R. Schindlauer, and H. Tompits. Reasoning with rules and ontologies. In P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler, editors, *Reasoning Web, Second International Summer School 2006, Lisbon, Portugal, September 4-8, 2006, Tutorial Lectures*, volume 4126 of *LNCS*, pages 93–127. Springer, September 2006.
7. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In L.P. Kaelbling and A. Saffiotti, editors, *IJCAI2005*, pages 90–96. Professional Book Center, 2005.
8. V. Haarslev and R. Möller. RACER system description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR 2001*, volume 2083 of *LNCS*, pages 701–706. Springer, 2001.
9. S. Heymans, T. Eiter, and G. Xiao. Tractable reasoning with DL-programs over Datalog-rewritable description logics. In H. Coelho, R. Studer, and M. Wooldridge, editors, *19th ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 35–40. IOS Press, 2010.
10. B. Motik and R. Rosati. Reconciling description logics and rules. *Journal of the ACM*, 57, June 2010. Article Nr 30.
11. R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3(1):61–73, 2005.