# Forgetting in Managing Rules and Ontologies*

Thomas Eiter[1], Giovambattista Ianni[1], Roman Schindlauer[1], Hans Tompits[1], and Kewen Wang[1,2]

[1] Institut für Informationssysteme 184/3, Technische Universität Wien, Favoritenstrasse 9-11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits, kewen}@kr.tuwien.ac.at
[2] School of Information and Communication Technology, Griffith University, Brisbane, QLD 4111, Australia

**Abstract.** The language of HEX-programs under the answer-set semantics is designed for interoperating with heterogeneous sources via external atoms and for meta-reasoning via higher-order literals in the context of the Semantic Web. As an important technique in managing knowledge bases, the notion of forgetting has received increasing interest in the knowledge-representation area. In this paper, we introduce a semantics-based theory of forgetting for HEX-programs and, in turn, for a class of OWL/RDF ontologies which allows to fully employ semantic information in managing ontologies like editing, merging, aligning, and redundancy removal.

## 1 Introduction

An ontology is a formal representation of concepts and relationships between them, making global interoperability possible. Managing ontologies is a central task for many Semantic-Web applications. However, it is often acknowledged that the Ontology Layer of the Semantic Web [1] is insufficient in its reasoning abilities. In particular, more and more ontologies are available on the Web and they are often very large in size and heterogeneous in location.

This phenomenon brings up a good deal of challenges to researchers in the Semantic Web. For example, when an ontology design is involved, we have to consider some issues like how to tailor an ontology or how to merge ontologies. Recently, these and related issues of managing ontologies have received considerable interests [12, 17, 18, 9, 10, 7]. Related issues include ontology editing, ontology segmentation, ontology merging, ontology aligning, ontology reusing, ontology update, and ontology redundancy removal. To some extent, all of these issues can be reduced to the problem of *extracting relevant segments out of large ontologies* for the purpose of effective management of ontologies so that the tractability for both humans and computers is enhanced. Such segments are not mere fragments of ontologies, but stand alone as ontologies in their own right. The intuition here is similar to views in databases: an existing ontology is

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

tailored to a smaller ontology so that an optimal ontology is produced for specific applications. Although this problem has been identified and a number of approaches are proposed, like, e.g., [8, 20], a general framework for tailoring ontologies in a purely semantic way is still missing.

On the other hand, the notion of forgetting [4, 15, 14] is a promising technique for adequately handling a range of classical tasks such as query answering, planning, decision-making, reasoning about actions, or knowledge update and revision. The idea of forgetting consists, informally, in the intelligent and "painless" removal of information from a given knowledge base. In other words, one may select some literals, predicates, or concepts, for being discarded (or *forgotten*) in a given knowledge base. However, the information selected for elimination is usually logically connected with other portions of the same knowledge base. It is thus important to preserve, to the best extent, soundness and completeness of the information entailed after removal.

This is similar in nature to the aforementioned problem in the design and engineering of Web-based ontology languages. Consider a scenario from [8]: Suppose we start to design an ontology about various pets (like cats or dogs, but not lions or tigers). As currently there are numerous ontologies on the Web, suppose we searched the Web and found a large ontology on various animals including cats, dogs, tigers and lions. It may not be appropriate to adopt and use the whole ontology. For example, we may wish to discard (or "forget") tigers and lions from it.

While a literature on forgetting in logic programming exists (see, e.g., [22, 4]), and although forgetting takes relevance also in ontology-description formalism such as OWL, an explicit notion of forgetting has not been given yet for this class of languages. In this respect, the relationship between a notion of forgetting in ontologies and of forgetting in rule-based formalisms has not satisfactorily been investigated yet, and is thus matter of new research.

The problem of forgetting in ontologies can indeed be solved by exploiting the connection between ontology-description formalisms and logic programming. That is, given a sound notion of forgetting for logic programming, a knowledge base $L$, formulated under a generic semantics (e.g. RDFS, OWL, etc.) can be transposed to an equivalent logic program $P_L$, formulated under a different (and usually, nonmonotonic) semantics. Then, logic programming forgetting techniques are applied to $P_L$ and a modified program, $\mathsf{forget}(P_L, l)$, is obtained and translated back to a knowledge base $L'$, where $l$ is the information to be discarded, which can be either a propositional atom, a concept, or a predicate.

Nonetheless, in order to fulfill the above approach, several issues, some of which already tackled in the literature, have to be solved and accommodated:

– A systematic way for translating $L$ to $P_L$ must be given. Attempts in this direction are several: for instance, Grosof *et al.* [11] translate a fragment of OWL-DL to Horn logic, whereas Swift [21] and Motik, Volz, and Maedche [16] port significant fragments of description logics to positive disjunctive logic programs.
– The pre-existing forgetting semantics [22, 4] mainly concentrates on discarding propositional information from ground programs. However, often $P_L$ might be a non-ground program and $l$ a non-propositional value (such as a predicate whose entire extension must be discarded). Also, many ontology description languages

(such as RDF and RDFS) include the possibility of exchanging the notion of class with the notion of individual, in order to enable meta-reasoning. In such a setting, $P_L$ is better mapped to a higher-order logic program.

- Also it is unclear in which cases $\mathsf{forget}(P_L, l)$ can be mapped back to a valid knowledge base $L'$.

In the present paper, we aim at answering some of the questions above.

The logic programming language of choice is HEX, as defined in previous work [3]. This is a rule-based, fully declarative formalism which allows both for higher-order atoms and external atoms, under a well-defined generalization of the answer-set semantics [6].

Intuitively, a higher-order atom allows to quantify values over predicate names and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) \leftarrow \mathit{subClassOf}(D, C), D(X).$$

An external atom facilitates the assignment of a truth value of an atom through an external source of computation. For instance, the rule

$$t(\mathit{Sub}, \mathit{Pred}, \mathit{Obj}) \leftarrow \&\mathit{rdf}[\mathit{uri}](\mathit{Sub}, \mathit{Pred}, \mathit{Obj})$$

computes the predicate $t$ taking values from the predicate $\&\mathit{rdf}$. The latter extracts RDF statements from the set of URIs specified by the extension of the predicate $\mathit{uri}$; this task is delegated to an external computational source (e.g., an external deduction system, an execution library, etc.). External atoms allow for a bidirectional flow of information to and from external sources of computation such as description-logic reasoners. By means of HEX-programs, powerful meta-reasoning becomes available in a decidable setting, e.g., not only for Semantic-Web applications, but also for meta-interpretation techniques in answer-set programming (ASP) itself, or for defining policy languages.

The contributions in this paper can be summarized as follows:

1. We introduce the notion of semantic forgetting for HEX-programs. Forgetting in logic programs has been previously considered by Eiter and Wang [4], who defined forgetting of a given literal $l$ in the context of propositional disjunctive logic programs. This notion is extended in order to deal with external and higher-order atoms, as well as with positive non-ground programs.
2. We develop an algorithm for forgetting which is useful in the setting of ontology management. The basic idea of this algorithm is that certain rules that are locally redundant may become relevant afterwards and thus they are kept in the program.
3. We show how semantic forgetting of ontologies can be performed using an equivalent logic program, whose modified versions (after forgetting) are translated back to ontologies. In particular, that fragment of OWL-DL is taken into account which can be translated to description-logic programs [11]. The approach can be currently generalized to all those ontology languages for which a sound and complete mapping to positive logic programs is known.

Our approach is illustrated on some example application. For this, we use an ontology “Person-Relationship” in the paper, which can be scaled as large as one wishes.

The rest of the paper is organized as follows. Section 2 briefly recalls syntax and semantics of HEX-programs. Section 3 introduces the notion of semantic forgetting for HEX-programs and a novel algorithm for computing forgetting. As well, forgetting for non-ground positive programs is defined. Section 4, then, discusses a method for forgetting OWL/RDF-ontologies in terms of a transformation technique. Finally, Section 5 wraps up the paper with some concluding remarks.

## 2 HEX-Programs

### 2.1 Syntax

HEX programs are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant names*, *variable names*, and *external predicate names*, respectively. Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are written with first letter in upper case (resp., lower case), and elements from $\mathcal{G}$ are prefixed with "&." Constant names serve both as individual and predicate names. Importantly, $\mathcal{C}$ may be infinite.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms and $n \geq 0$ is its *arity*. Intuitively, $Y_0$ is the predicate name; we thus also use the familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if $Y_0$ is a constant. For example, $(x, \mathit{rdf}{:}\mathit{type}, c)$ and $\mathit{node}(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom. An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{1}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input list* and *output list*, respectively), and $\&g$ is an *external predicate name*.

It is possible to specify *molecules* of atoms similar as in F-Logic [13]. For instance, $gi[\mathit{father} \rightarrow X, Z \rightarrow iu]$ is a shortcut for the conjunction $\mathit{father}(gi, X)$, $Z(gi, iu)$.

A HEX-*program*[1] is a set of rules of the form

$$\alpha \leftarrow \beta_1, \ldots, \beta_n, \mathit{not}\ \beta_{n+1}, \ldots, \mathit{not}\ \beta_m, \tag{2}$$

where $m \geq 0$, $\alpha$ is a higher-order atom, and $\beta_1, \ldots, \beta_m$ are either higher-order atoms or external atoms. The operator "*not*" is *negation as failure* (or *default negation*). For a rule $r$ as in (2), we define $\mathit{head}(r) = \alpha$ and $\mathit{body}(r) = \mathit{body}^+(r) \cup \mathit{body}^-(r)$, where $\mathit{body}^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $\mathit{body}^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. If $r$ contains only ordinary atoms, then $r$ is *ordinary*. Furthermore, $r$ is *quasi-negative* if $n = 0$. A HEX-program is *quasi-negative* if it contains only quasi-negative rules. An ordinary rule is *positive* iff $m = n$, i.e., if it contains no negation as failure. A program is positive iff all rules in it are positive.

We mention that higher-order features in logic programs have also been considered, e.g., by Chen, Kifer, and Warren [2] and Ross [19].

[1] In contrast to the original definition in [3], here we consider only HEX-programs without disjunctions in rule heads.

### 2.2 Semantics

The semantics of HEX-programs [3] is defined by generalizing the answer-set semantics [6]. The *Herbrand base* of a program $P$, denoted $HB_P$, is the set of all possible ground versions of atoms and external atoms occurring in $P$ obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule $r$, $grnd(r)$, is defined accordingly, and the grounding of program $P$ is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ are implicitly given by $P$.

An *interpretation relative to* $P$ is any subset $I \subseteq HB_P$ containing only atoms. We say that an interpretation $I$ is a *model* of an atom $a \in HB_P$ iff $a \in I$. Furthermore, $I$ is a model of a ground external atom $a = \&g[y_1, \dots, y_n](x_1, \dots, x_m)$ iff $f_{\&g}(I, y_1, \dots, y_n, x_1, \dots, x_m) = 1$, where $f_{\&g}$ is an $(n{+}m{+}1)$-ary Boolean function associated with $\&g$, called *oracle function*, assigning each element of $2^{HB_P} \times \mathcal{C}^{n+m}$ either $0$ or $1$. We write $I \models a$ to express that $I$ is a model of $a$.

Let $r$ be a ground rule. We define (i) $I \models body(r)$ iff $I \models a$ for all $a \in body^+(r)$ and $I \not\models a$ for all $a \in body^-(r)$, and (ii) $I \models r$ iff $I \models head(r)$ whenever $I \models body(r)$. We say that $I$ is a *model* of a HEX-program $P$, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *Faber-Leone-Pfeifer reduct* [5] (or short *FLP-reduct*) of $P$ with respect to $I \subseteq HB_P$, denoted $fP^I$, is the set of all $r \in grnd(P)$ such that $I \models body(r)$. $I \subseteq HB_P$ is an *answer set of* $P$ iff $I$ is a minimal model of $fP^I$. By $\mathsf{AS}(P)$ we denote the set of all answer sets of $P$.

A HEX-program is *consistent* if it has at least one answer set. We call two HEX-programs, $P$ and $Q$, *equivalent*, symbolically $P \equiv Q$, iff $\mathsf{AS}(P) = \mathsf{AS}(Q)$.

In practice, it is useful to differentiate between two kinds of input attributes for external atoms. For an external predicate $\&g$ (exploited, say, in an atom $\&g[p](X)$), a term appearing in an attribute position of type *predicate* (in this case, $p$) means that the outcomes of $f_{\&g}$ are dependent from the current interpretation $I$, for what the extension of the predicate named $p$ in $I$ is concerned. An input attribute of type *constant* does not imply a dependency of $f_{\&g}$ from some portion of $I$. An external predicate whose input attributes are all of type constant does not depend from the current interpretation.

*Example 2.1.* The external predicate $\&rdf$ introduced before is implemented with a single input argument of type predicate, because its associated function finds the RDF-URIs in the extension of the predicate $uri$:

$$\begin{aligned} tr(S,P,O) &\leftarrow \&rdf[uri](S,P,O), \\ uri(\text{“}\mathit{file://foaf.rdf}\text{”}) &\leftarrow . \end{aligned}$$

Should the input argument be of type constant, an equivalent program would be:

$$tr(S,P,O) \leftarrow \&rdf[\text{“}\mathit{file://foaf.rdf}\text{”}](S,P,O)$$

or

$$\begin{aligned} tr(S,P,O) &\leftarrow \&rdf[X](S,P,O), uri(X), \\ uri(\text{“}\mathit{file://foaf.rdf}\text{”}) &\leftarrow . \end{aligned}$$

□

## 3 Forgetting in HEX-Programs

As we have explained in Section 1, the technique of forgetting is useful in managing ontologies. So it is natural and interesting to generalize forgetting to HEX-programs. In fact, since HEX-programs have higher-order syntax but first-order semantics, it allows us to adapt the notion of forgetting to HEX-programs. In this section, we introduce the notion of forgetting for HEX-programs. The intuition behind the forgetting of an atom $l$ in a HEX-program is to obtain a HEX-program which is equivalent to the original HEX-program if we ignore the existence of $l$.

In the next subsection, we assume that HEX-programs are ground and consistent. When a HEX-program with variables is given, it is a shorthand for its ground version. As we will see in Section 4, forgetting in an RDF ontology is defined in terms of forgetting in the corresponding logic program, which is a non-ground positive program. So, in Subsection 3.2, forgetting in non-grounded positive programs is considered.

### 3.1 Forgetting in Ground HEX-Programs

We call a set $X'$ an *$l$-subset* of a set $X$, denoted $X' \subseteq_l X$, if $X' \setminus \{l\} \subseteq X \setminus \{l\}$. Similarly, a set $X'$ is a *strict $l$-subset* of $X$, denoted $X' \subset_l X$, if $X' \setminus \{l\} \subset X \setminus \{l\}$. Two sets $X$ and $X'$ of literals are *$l$-equivalent*, denoted $X \sim_l X'$, if $(X \setminus X') \cup (X' \setminus X) \subseteq \{l\}$.

**Definition 3.1.** *Let $P$ be a consistent* HEX*-program, let $l$ be a (ground) atom in $P$, and let $X$ be a set of atoms.*

1. *For a collection $\mathcal{S}$ of sets of atoms, $X \in \mathcal{S}$ is $l$-*minimal *in $\mathcal{S}$ if there is no $X' \in \mathcal{S}$ such that $X' \subset_l X$.*
2. *An answer set $X$ of a* HEX*-program $P$ is an $l$-*answer set *if $X$ is $l$-minimal in* $\mathsf{AS}(P)$.

*Example 3.1.* Let $P = \{p \leftarrow not\, q;\ q \leftarrow not\, p;\ s \leftarrow p;\ s \leftarrow q\}$. It is easy to see that $P$ has two answer sets, viz. $X = \{p, s\}$ and $X' = \{q, s\}$. Then, $X$ is a $p$-answer set but $X'$ is not. □

Having defined the notion of minimality about forgetting an atom, we are now in a position to define the result of forgetting about an atom in a HEX-program.

**Definition 3.2.** *Let $P$ be a consistent* HEX*-program and let $l$ be a (ground) atom. A* HEX*-program $P'$ is a* result of forgetting about $l$ in $P$, *if $P'$ represents $l$-answer sets of $P$, i.e., such that the following conditions are satisfied:*

1. *$At(P') \subseteq At(P) - \{l\}$, where, for any program $Q$, $At(Q)$ denotes the set of atoms occurring in $Q$.*
2. *For any set $X'$ of atoms with $l \notin X'$, $X'$ is an answer set of $P'$ iff there is an $l$-answer set $X$ of $P$ such that $X' \sim_l X$.*

Note that the first condition implies that $l$ does not appear in $P'$. We use $\mathsf{forget}(P, l)$ to denote a possible result of forgetting about $l$ in $P$.

Since an atom that does not appear in the head of a rule in a HEX-program is automatically assumed to be false in the process of forgetting for ordinary programs, all external atoms would be removed from the program. For this reason, the native algorithm for forgetting [4] is not helpful for HEX-programs. Thus, we introduce a new algorithm, which is inspired by Algorithm 4 in the system LPForget.[2]

Preparatory for describing the algorithm, below we introduce some program transformations for HEX-programs, which are generalizations of respective ones for ordinary programs [4].

In the following, let $P$ and $P'$ be HEX-programs.

**Elimination of Tautologies:** $P'$ is obtained from $P$ by *elimination of tautologies* iff there is a rule $r$ in $P$ such that $head(r) \in body^{+}(r)$ and $P' = P - \{r\}$.

**Elimination of Head Redundancy:** $P'$ is obtained from $P$ by *elimination of head redundancy* iff there is a rule $r$ in $P$ such that $head(r) \in body^{-}(r)$ and $P' = (P - \{r\}) \cup \{\leftarrow body(r)\}$.

**Positive Reduction:** $P'$ is obtained from $P$ by *positive reduction* iff there is a rule $r$ in $P$ such that $body^{-}(r)$ contains some $c$ which does not occur in the head of any rule in $P$ and $P'$ is obtained from $P$ by removing $not\ c$ from $r$.

**Negative Reduction:** $P'$ is obtained from $P$ by *negative reduction* iff there are two rules $r$ and $r' : b' \leftarrow$ in $P$ such that $b' \in body^{-}(r)$ and $P' = P - \{r\}$.

**Elimination of Implications:** Let $r$ and $r'$ be two distinct rules in a logic program. We say that $r'$ is an *implication* of $r$ if $head(r) = head(r')$ and $body(r) \subset body(r')$. Then, $P'$ is obtained from $P$ by *elimination of implications* iff there are two distinct rules $r$ and $r'$ of $P$ such that $r'$ is an implication of $r$ and $P' = P - \{r'\}$.

**Unfolding:** For two rules $r$ and $r'$ with $head(r') \in body^{+}(r)$, the *unfolding* of $r$ with $r'$, denoted $unfold(r, r')$, is the rule $head(r) \leftarrow (body(r) - \{head(r')\}), body(r')$. Then, $P'$ is obtained from $P$ by *unfolding* if there is a rule $r$ such that

$$P' = (P - \{r\}) \cup \{unfold(r, r') \mid r' \in P, head(r') \in body^{+}(r)\}.$$

A special case of unfolding is when there is no rule $r'$ such that $r'$ is resolved with $r$. In this case, $P' = P - \{r\}$.

We use $\mathcal{T}$ to denote the set of program transformations introduced above.

**Lemma 3.1.** *Using program transformations in $\mathcal{T}$, every* HEX*-program can be transformed into a quasi-negative program such that no atom appears in both head and body of a rule.*

The algorithm for computing the result of forgetting, referred to as *Algorithm 1*, is depicted in Figure 1. This algorithm can be easily implemented using the system LPForget. Note that the current form of Algorithm 1 is incomplete with respect to the semantic forgetting for some special cases while it is intuitive and can be seen an

[2] See `http://www.cit.gu.edu.au/~kewen/LPForget/`.

**Algorithm 1 (Computing a result of forgetting)**
**Input**: HEX-program $P$ and an atom $l$ in $P$.
**Output**: Program $\mathsf{forget}(P, l)$ as a result of forgetting $l$ from $P$.
**Method:**
*Step 1.* Positive Splitting: Initially take $Q$ as the set of all rules in which $l$ appears. For every rule $r$ in $P$ such that either $head(r)$ or some literal of $body^-(r)$ appears in $Q$, add $r$ to $Q$. Repeat this process until no new rule can be added. The resulting program is still denoted $Q$.
*Step 2.* Fully apply on $Q$ the program transformations $\mathcal{T}$ and then obtain a quasi-negative program $Q'$. During this process, we keep record of the set $RU(Q, l)$ of all rules removed by unfolding but containing no appearance of $l$.
*Step 3.* Suppose that $Q'$ has $n$ rules with head $l$:

$$r_j : l \leftarrow not\, l_{j1}, ..., not\, l_{jm_j},$$

where $n \geq 0$, $j = 1, \ldots, n$ and $m_j \geq 0$ for all $j$.
If $n = 0$, then let $Q''$ denote the program obtained from $Q'$ by removing all appearances of $not\, l$.
If $n = 1$ and $m_1 = 0$, then $l \leftarrow$ is the only rule in $Q'$ having head $l$. In this case, remove every rule in $Q'$ whose body contains $not\, l$. Let $Q''$ be the resulting program.
For $n \geq 1$ and $m_1 > 0$, let $D_1, \ldots, D_s$ be all possible conjunctions $(l_{1k_1}, \cdots, l_{nk_n})$, where $0 \leq k_1 \leq m_1$, ..., $0 \leq k_n \leq m_n$. Replace each occurrence of $not\, l$ in $Q'$ by all possible $D_i$. Let $Q''$ be the result.
*Step 4.* Output $Q'' \cup RU(Q, l) \cup \bar{Q}$ as $\mathsf{forget}(P, l)$, where $\bar{Q} = P \setminus Q$.

**Fig. 1.** Algorithm 1 for computing a result of forgetting.

ideal approximation to the semantic forgetting. A complete algorithm is obtained by replacing Step 3 with Step 3 of Algorithm 2 given by Eiter and Wang [4].

For a consistent HEX-program $P$ and an atom $l$, some program $P'$ as in Definition 3.2 always exists. However, different such programs $P'$ might exist. It follows from the above definition that they are all equivalent under the answer-set semantics.

**Proposition 3.1.** *Let $P$ be a HEX-program and $l$ an atom in $P$. If $P'$ and $P''$ are two results of forgetting about $l$ in $P$, then $P' \equiv P''$.*

*Example 3.2.* Suppose that $L$ is a knowledge base on the Web consisting of various axioms about persons and their relationships. In particular, $L$ contains assertions depicted in Figure 2.

Let $P$ now be the following HEX-program, where $\&dlC$ and $\&dlR$ are external atoms that query the extensions of a specified concept resp. role from a single description logic ontology:[3]

$$\begin{aligned} sibling(X, Y) &\leftarrow \&dlR[siblingOf](X, Y); \\ sibling(X, Y) &\leftarrow \&dlR[childOf](X, Z), \&dlR[childOf](Y, Z); \\ inEurope(Y) &\leftarrow sibling(\text{“}Bob\text{”}, Y), not\ inAmerica(Y); \\ inAmerica(Y) &\leftarrow sibling(\text{“}Bob\text{”}, Y), not\ inEurope(Y). \end{aligned}$$

[3] For the sake of readability, we use a simplified version of the actual and implemented dl-atoms for HEX-programs here.

$$\begin{array}{ll}
\mathit{Male} \sqsubseteq \mathit{Person} & \mathit{spouseOf} \sqsubseteq \mathit{knows} \\
\mathit{Female} \sqsubseteq \mathit{Person} & \mathit{spouseOf} \equiv \mathit{spouseOf}^{-} \\
\top \sqsubseteq \forall \mathit{knows}^{-}.\mathit{Person} & \mathit{worksWith} \equiv \mathit{worksWith}^{-} \\
\top \sqsubseteq \forall \mathit{knows}.\mathit{Person} & \mathit{worksWith}^{+} \sqsubseteq \mathit{worksWith} \\
\mathit{friendOf} \sqsubseteq \mathit{knows} & \mathit{worksWith} \sqsubseteq \mathit{knows} \\
\mathit{childOf} \sqsubseteq \mathit{knows} & \mathit{parentOf} \equiv \mathit{childOf}^{-} \\
\mathit{siblingOf} \equiv \mathit{siblingOf}^{-} & \mathit{parentOf} \sqsubseteq \mathit{ancestorOf} \\
\mathit{siblingOf}^{+} \sqsubseteq \mathit{siblingOf} & \mathit{ancestorOf} \sqsubseteq \mathit{knows} \\
\mathit{siblingOf} \sqsubseteq \mathit{knows} & \mathit{ancestorOf}^{+} \sqsubseteq \mathit{ancestorOf} \\
\mathit{parentOf}(\mathit{Alice}, \mathit{Bob}) & \mathit{sameProject} \sqsubseteq \mathit{worksWith} \\
\mathit{parentOf}(\mathit{Alice}, \mathit{Carl}) & \\
\mathit{parentOf}(\mathit{Bob}, \mathit{Emma}) & \\
\mathit{sameProject}(\mathit{Bob}, \mathit{Dennis}) &
\end{array}$$

**Fig. 2.** Example ontology $L$.

To apply forgetting, we first have to obtain the ground program $\mathit{grnd}(P)$. In order to keep the example readable, we omit those ground rules whose bodies are not satisfied by $L$:

$$\begin{aligned}
\mathit{sibling}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Carl}\text{”}) &\leftarrow \&\mathit{dlR}[\mathit{childOf}](\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Alice}\text{”}), \\
&\phantom{\leftarrow} \&\mathit{dlR}[\mathit{childOf}](\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Alice}\text{”}); \\
\mathit{sibling}(\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Bob}\text{”}) &\leftarrow \&\mathit{dlR}[\mathit{childOf}](\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Alice}\text{”}), \\
&\phantom{\leftarrow} \&\mathit{dlR}[\mathit{childOf}](\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Alice}\text{”}); \\
\mathit{inEurope}(\text{“}\mathit{Carl}\text{”}) &\leftarrow \mathit{sibling}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Carl}\text{”}), \mathit{not}\ \mathit{inAmerica}(\text{“}\mathit{Carl}\text{”}); \\
\mathit{inAmerica}(\text{“}\mathit{Carl}\text{”}) &\leftarrow \mathit{sibling}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Carl}\text{”}), \mathit{not}\ \mathit{inEurope}(\text{“}\mathit{Carl}\text{”}).
\end{aligned}$$

Thus, $\mathit{grnd}(P)$ has two answer sets, viz.

$$X_1 = \{\mathit{sibling}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Carl}\text{”}), \mathit{sibling}(\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Bob}\text{”}), \mathit{inEurope}(\text{“}\mathit{Carl}\text{”})\} \text{ and}$$
$$X_2 = \{\mathit{sibling}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Carl}\text{”}), \mathit{sibling}(\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Bob}\text{”}), \mathit{inAmerica}(\text{“}\mathit{Carl}\text{”})\}.$$

If we allow to forget about $\mathit{sibling}(\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Bob}\text{”})$ in $\mathit{grnd}(P)$, then the result of forgetting is obtained from $\mathit{grnd}(P)$ by removing the first rule. □

The above definitions of forgetting about an atom $l$ can be extended to forgetting about a set $F$ of atoms. Specifically, we can similarly define $X_1 \subseteq_F X_2$, $X_1 \sim_F X_2$, and $F$-answer sets of a HEX-program. In fact, the properties of forgetting about a single atom can be generalized to the case of forgetting about a set. Moreover, the result of forgetting about a set $F$ can be obtained by forgetting each atom one by one in $F$.

**Proposition 3.2.** *Let $P$ be a consistent* HEX*-program and $F = \{l_1, \ldots, l_m\}$ a set of atoms. Then,* $\mathsf{forget}(P, F) \equiv \mathsf{forget}(\ldots(\mathsf{forget}(\mathsf{forget}(P, l_1), l_2), \ldots), l_m)$.

Since higher-order atoms and external atoms can be treated as ordinary atoms in the process of forgetting, we can prove the above result similarly to the proof of Proposition 6 given by Eiter and Wang [4].

For HEX-programs, the notion of ordinary forgetting may not be sufficient for some applications in managing ontologies. In some cases, we need to forget a *predicate*. This can be easily accomplished by forgetting the set of all atoms with the same predicate.

Due to the presence of higher-order terms, we may need also to forget some other atoms when we want to forget a specific atom. This is illustrated in the following example.

*Example 3.3.* Suppose we want to forget the predicate "$brotherOf$" in the following program:

$$\begin{aligned}
subRelation(brotherOf, siblingOf) &\leftarrow \\
brotherOf(john, al) &\leftarrow \\
siblingOf(john, joe) &\leftarrow \\
siblingOf(al, mick) &\leftarrow \\
R(X, Y) &\leftarrow subRelation(P, R), P(X, Y)
\end{aligned}$$

Here, we should also forget $subRelation(brotherOf, siblingOf)$. □

For the above discussion, it is natural to define the following variant of forgetting, which is more intuitive for most applications.

**Definition 3.3.** *Let $P$ be a* HEX*-program and $l$ an atom in $P$. Denote by $sup(l)$ the set of all atoms in $P$ that contain the predicate name of $l$. Then the result of* enforced forgetting *about $l$ in $P$, written* $\mathsf{Forget}(P, l)$*, is defined as* $\mathsf{forget}(P, sup(l))$.

In Example 3.3, $\mathsf{Forget}(P, brotherOf)$, given by

$$\mathsf{forget}(P, \{brotherOf(john, al), subRelation(brotherOf, siblingOf)\}),$$

is the following program:

$$\begin{aligned}
siblingOf(john, al) &\leftarrow ; \\
siblingOf(john, joe) &\leftarrow ; \\
siblingOf(al, mick) &\leftarrow .
\end{aligned}$$

## 3.2 Forgetting in Non-Ground Positive Programs

As we will see in Section 4, the logic program $P_L$ translated from an OWL/RDF ontology is non-ground in general and thus forgetting as defined by Eiter and Wang [4] cannot be directly applied here. However, since $P_L$ has a special form and, in particular, has no negation as failure, we are able to lift the notion of forgetting for ground programs to this kind of non-ground programs.

To this end, we first need to define *weak unfolding* for logic programs.

Let $r : a \leftarrow b, B$ and $r' : b' \leftarrow B'$ be normal rules, where $a, b, b'$ are atoms, and $B$, $B'$ are conjunctions of literals. Note that no higher-order atoms occur here. When necessary, we can rename the variables of $r'$ such that $r$ and $r'$ have no common variables. If the head $b'$ of $r'$ and $b$ have a most general unifier (mgu) $\theta$, then the rule $(a \leftarrow B, B')\theta$ is called a *resolvent* of $r$ with $r'$.

**Algorithm 2 (Computing forgetting for non-ground positive logic programs)**
**Input**: Positive logic program $P$ and a predicate $R$.
**Output**: Program $\mathsf{forget}(P, R)$ as the result of forgetting $R$ from $P$.
**Method**:

1. Fully apply weak unfolding on $P$.
2. Remove all rules containing $R$.
3. Output the resulting program as $\mathsf{forget}(P, R)$.

**Fig. 3.** Computing forgetting for non-ground programs without negation as failure.

**Weak Unfolding.** A logic program $P'$ is obtained from $P$ by *weak unfolding* iff there are two rules $r$ and $r'$ in $P$ such that $r''$ is a resolvent of $r$ with $r'$ and $P' = P \cup \{r''\}$.

For a positive logic program the result of forgetting can be easily obtained by Algorithm 2 depicted in Figure 3.

The following result shows that this lifting algorithm for forgetting is sound with respect to semantic forgetting for ground programs.

**Theorem 3.1.** *Let $P$ be a non-ground positive program and $R$ a predicate in $P$. For any extensional database $E$ (i.e., a set of facts), we have*

$$\mathsf{forget}(P, R) \cup E \equiv \mathsf{forget}(grnd(P \cup E), const(R)),$$

*where $const(R) = \{R(a) \mid a \text{ is a constant in } P \cup E\}$.*

*Proof.* (Sketch) First, we observe the following two properties:

($\alpha$) If $r''$ is an instance of $unfold(r, r')$ in $P \cup E$, then $r'' = unfold(\bar{r}, \bar{r}')$, where $\bar{r}$ and $\bar{r}'$ are instances of $r$ and $r'$, respectively.
($\beta$) If $r'' = unfold(\bar{r}, \bar{r}')$, for $\bar{r}$ and $\bar{r}'$ in $grnd(P \cup E)$, then $r''$ is an instance of $unfold(r, r')$ in $P \cup E$, where $\bar{r}$ and $\bar{r}'$ are instances of $r$ and $r'$, respectively.

Let $Q_1 = \mathsf{forget}(P, R) \cup E$ and $Q_2 = \mathsf{forget}(grnd(P \cup E), const(R))$. Since $P$ is positive, $\mathsf{AS}(grnd(Q_1))$ and $\mathsf{AS}(Q_2)$ are singletons.

Let $T_Q$ be the consequence operator of a positive program $Q$. Then, the unique answer set of $Q$ is its least Herbrand model $\bigcup_{k \geq 0} T_Q \uparrow k$.

The unique element of $\mathsf{AS}(grnd(Q_1))$ is $\cup_{k \geq 0} T_{grnd(Q_1)} \uparrow k$, and the unique element of $\mathsf{AS}(Q_2)$ is $\bigcup_{k \geq 0} T_{Q_2} \uparrow k$. Using Properties ($\alpha$) and ($\beta$), we can easily show that $T_{grnd(Q_1)} \uparrow k{=}T_{Q_2} \uparrow k$, by induction on $k \geq 0$. So, $\mathsf{AS}(grnd(Q_1)){=}\mathsf{AS}(Q_2)$. $\square$

Algorithm 2 may be refined by applying Step 1 only to a subset of the rules and facts $P$ which is relevant to $R$, while the rest of the program remains untouched. In this way, the cost of computing forgetting can be reduced radically.

For $P$ and $R$ in Algorithm 2, let $Q$ initially be the set of all rules in which $R$ appears. Then, add each rule $r$ from $P$ to $Q$ such that $head(r)$ appears in $Q$, and repeat

$$\begin{tikzcd}
L \arrow[r] & P_L \arrow[d] \\
L' & \mathsf{forget}(P_L, l) \arrow[l]
\end{tikzcd}$$

**Fig. 4.** Forgetting in ontologies via a logic program.

this process until no new rules can be added. Let the resulting program be denoted by $Q_{P,R}$. Intuitively, $\bar{Q}$ consists of rules that are irrelevant to $R$. For forgetting in a positive program, it is done by a series of unfolding and then removing some rules relevant to $R$. So, rules in $\bar{Q}$ are essentially unchanged during the process of forgetting.

**Theorem 3.2.** *Let $P$ be a non-ground positive program $P$ and let $R$ be a predicate in $P$. For any extensional database $E$, it holds that*

$$\mathsf{forget}(P, R) \cup E \equiv \mathsf{forget}(Q_{P,R}, R) \cup \bar{Q} \cup E,$$

*where $\bar{Q} = P \setminus Q_{P,R}$.*

It should be noted that although the process of forgetting for non-ground programs is realized by the removal of certain rules, it has a semantic justification as Theorem 3.2 shows.

## 4 Forgetting in OWL/RDF-Ontologies

To apply forgetting purely to an ontology expressed in OWL or RDFS, we reuse the techniques defined for forgetting in logic programs. Figure 4 shows the general principle of this approach. First, an ontology $L$ is translated into a rule representation $P_L$, taking the specific ontology semantics into account. Then, for any atom $l$ in $P_L$, we can compute $\mathsf{forget}(P_L, l)$. Finally, we translate the result back into an ontology.

The translation of description-logic axioms into a logic program is shown in Tables 1 and 2. This translation covers most of the expressiveness of OWL Lite and corresponds to the translation given by Grosof *et al.* [11], mapping some subset of a description logic to positive equality-free datalog programs. Note that some description-logic constructs have no direct representation in logic-programming rules, such as cardinality constraints. Also, existential and universal quantification is restricted to the left-hand side resp. right-hand side of a subclass axiom. In general, a transformation from a set of rules back to ontology statements requires the rules in $\mathsf{forget}(P_L, l)$ to be in a form according to Tables 1 and 2.

*Example 4.1.* Consider again the ontology $L$ in Figure 2. The translation of $L$ into a logic program according to $P_L$ is depicted in Figure 5. Suppose we do not want to keep the concepts $worksWith$, then we can use Theorem 3.1 to simplify the process of forgetting.

Take $Q$ as a subprogram of $P_L$:

**Table 1.** Mapping of ontology statements to rules.

| **Statement** | **DL syntax** | **Rule representation** |
|---|---|---|
| subClassOf | $D \sqsubseteq C$ | $C(X) \leftarrow D(X).$ |
| subPropertyOf | $P \sqsubseteq Q$ | $Q(X,Y) \leftarrow P(X,Y).$ |
| domain | $\top \sqsubseteq \forall P^{-}.C$ | $C(X) \leftarrow P(X,Y).$ |
| range | $\top \sqsubseteq \forall P.C$ | $C(Y) \leftarrow P(X,Y).$ |
| class-instance | $a : C$ | $C(a) \leftarrow .$ |
| property-instance | $\langle a,b\rangle : P$ | $P(a,b) \leftarrow .$ |
| class-equivalence | $D \equiv C$ | $D(X) \leftarrow C(X);$ |
| | | $C(X) \leftarrow D(X).$ |
| property-equivalence | $P \equiv Q$ | $P(X,Y) \leftarrow Q(X,Y);$ |
| | | $Q(X,Y) \leftarrow P(X,Y).$ |
| inverseOf | $P \equiv Q^{-}$ | $P(X,Y) \leftarrow Q(Y,X);$ |
| | | $Q(X,Y) \leftarrow P(Y,X).$ |
| transitiveProperty | $P^{+} \sqsubseteq P$ | $P(X,Y) \leftarrow P(X,Z), P(Z,Y).$ |

**Table 2.** Mapping of ontology class constructors to rules.

| **Constructor** | **DL syntax** | **Rule representation** |
|---|---|---|
| conjunction | $C_1 \sqcap C_2 \sqsubseteq D$ | $D(X) \leftarrow C_1(X), C_2(X).$ |
| | $C \sqsubseteq D_1 \sqcap D_2$ | $D_1(X) \leftarrow C(X);$ |
| | | $D_2(X) \leftarrow C(X).$ |
| disjunction | $C_1 \sqcup C_2 \sqsubseteq D$ | $D(X) \leftarrow C_1(X);$ |
| | | $D(X) \leftarrow C_2(X).$ |
| existential restriction | $\exists P.C \sqsubseteq D$ | $D(X) \leftarrow P(X,Y), C(Y).$ |
| universal restriction | $D \sqsubseteq \forall P.C$ | $C(Y) \leftarrow P(X,Y), D(X).$ |

$$\begin{aligned}
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
worksWith(X,Y) &\leftarrow worksWith(Y,X);\\
worksWith(X,Z) &\leftarrow worksWith(X,Y), worksWith(Y,Z);\\
knows(X,Y) &\leftarrow worksWith(X,Y);\\
worksWith(X,Y) &\leftarrow sameProject(X,Y).
\end{aligned}$$

We can apply Algorithm 2 on the logic program $Q$ by forgetting $worksWith$. First, fully apply weak unfolding on $Q$ and obtain $Q'$:

$$\begin{aligned}
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
worksWith(X,Y) &\leftarrow worksWith(Y,X);\\
worksWith(X,Z) &\leftarrow worksWith(X,Y), worksWith(Y,Z);\\
knows(X,Y) &\leftarrow worksWith(X,Y);\\
worksWith(X,Y) &\leftarrow sameProject(X,Y);\\
knows(X,Y) &\leftarrow sameProject(X,Y);\\
worksWith(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
knows(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow .
\end{aligned}$$

$$\begin{aligned}
\mathit{parentOf}(\text{“}\mathit{Alice}\text{”}, \text{“}\mathit{Carl}\text{”}) &\leftarrow .\\
\mathit{female}(\text{“}\mathit{Alice}\text{”}) &\leftarrow .\\
\mathit{sameProject}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Dennis}\text{”}) &\leftarrow .\\
\mathit{male}(\text{“}\mathit{Bob}\text{”}) &\leftarrow .\\
\mathit{parentOf}(\text{“}\mathit{Carl}\text{”}, \text{“}\mathit{Emma}\text{”}) &\leftarrow .\\
\mathit{person}(\text{“}\mathit{Carl}\text{”}) &\leftarrow .\\
\mathit{person}(\text{“}\mathit{Dennis}\text{”}) &\leftarrow .\\
\mathit{knows}(X,Y) &\leftarrow \mathit{childOf}(X,Y).\\
\mathit{childOf}(X,Y) &\leftarrow \mathit{parentOf}(Y,X).\\
\mathit{male}(X) &\leftarrow \mathit{father}(X).\\
\mathit{person}(X) &\leftarrow \mathit{female}(X).\\
\mathit{friendOf}(X,Y) &\leftarrow \mathit{friendOf}(Y,X).\\
\mathit{knows}(X,Y) &\leftarrow \mathit{ancestorOf}(X,Y).\\
\mathit{ancestorOf}(X,Z) &\leftarrow \mathit{ancestorOf}(X,Y),\\
&\qquad \mathit{ancestorOf}(Y,Z).\\
\mathit{knows}(X,Y) &\leftarrow \mathit{friendOf}(X,Y).
\end{aligned}
\qquad
\begin{aligned}
\mathit{person}(X) &\leftarrow \mathit{knows}(X,Y).\\
\mathit{person}(Y) &\leftarrow \mathit{knows}(X,Y).\\
\mathit{person}(X) &\leftarrow \mathit{male}(X).\\
\mathit{female}(X) &\leftarrow \mathit{mother}(X).\\
\mathit{ancestorOf}(X,Y) &\leftarrow \mathit{parentOf}(X,Y).\\
\mathit{parentOf}(X,Y) &\leftarrow \mathit{childOf}(Y,X).\\
\mathit{siblingOf}(X,Y) &\leftarrow \mathit{siblingOf}(Y,X).\\
\mathit{knows}(X,Y) &\leftarrow \mathit{siblingOf}(X,Y).\\
\mathit{spouseOf}(X,Y) &\leftarrow \mathit{spouseOf}(Y,X).\\
\mathit{knows}(X,Y) &\leftarrow \mathit{spouseOf}(X,Y).\\
\mathit{worksWith}(X,Y) &\leftarrow \mathit{worksWith}(Y,X).\\
\mathit{worksWith}(X,Z) &\leftarrow \mathit{worksWith}(X,Y),\\
&\qquad \mathit{worksWith}(Y,Z).\\
\mathit{knows}(X,Y) &\leftarrow \mathit{worksWith}(X,Y).\\
\mathit{worksWith}(X,Y) &\leftarrow \mathit{sameProject}(X,Y).
\end{aligned}$$

**Fig. 5.** Translation of $L$ into a logic program $P_L$.

Thus, the result of forgetting about $\mathit{worksWith}$ is the program

$$\mathsf{forget}(Q, \mathit{worksWith}) \cup \bar{Q},$$

where $\bar{Q} = P_L \setminus Q$ and $\mathsf{forget}(Q, \mathit{worksWith})$ is as follows:

$$\begin{aligned}
\mathit{sameProject}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Dennis}\text{”}) &\leftarrow ;\\
\mathit{knows}(X,Y) &\leftarrow \mathit{sameProject}(X,Y);\\
\mathit{knows}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Dennis}\text{”}) &\leftarrow .
\end{aligned}$$

Translating this fragment back into the original description logic results in the following statements:

$\mathit{sameProject}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Dennis}\text{”})$;
$\mathit{sameProject} \sqsubseteq \mathit{knows}$;
$\mathit{knows}(\text{“}\mathit{Bob}\text{”}, \text{“}\mathit{Dennis}\text{”})$.

The property $\mathit{worksWith}$ does not occur in the modified description-logic knowledge base any more, while the subproperty relation between $\mathit{sameProject}$ and $\mathit{knows}$ is preserved. $\square$

Combining the approaches to forgetting of Sections 3 and 4, we are now able to forget any set of ordinary atoms, higher-order atoms, whole external atoms, and parts of external atoms in a HEX-program.

## 5 Related Work and Concluding Remarks

The notion of forgetting for HEX-programs introduced in this paper generalizes a respective notion for ordinary logic programs defined in previous work [4]. Forgetting for HEX-programs provides a means to handle forgetting at the user-view level, since HEX-programs are tailored to access sources like OWL/RDF ontologies at the extensional level through external atoms, but does not go back to changes in these sources, as is done in the view-update problem of databases, for instance. However, such ontologies have been cast to a class of logic programs which constitute a small fragment of HEX-programs, and thus semantic forgetting for OWL/RDF may be facilitated through this mapping, as we have shown. Our work therefore provides a uniform basis for a framework for extracting ontology segments from a custom ontology, which is exploited at the user level. This approach is in an active area of *semantic integration* in ontologies (see [17] for a survey). However, the emphasis of our work is on conflict resolving in semantic integration of ontologies rather than on ontology mapping.

Forgetting for OWL/RDF ontologies can be used for various tasks in ontology management including the following:

- *Ontology segmentation:* This approach is to obtain segments from a custom ontology, thus having the same purpose as forgetting. Seidenberg and Rector [20] present a series of strategies for extracting ontology segments. However, it lacks a general semantic justification.
- *Ontology merging:* Given two ontologies $O_1$ and $O_2$, they could first be preprocessed by techniques in ontology mapping and then be merged into one ontology. In many cases, conflicts may be present in the process of merging. If the conflict is caused by some concept $C$, a natural approach is to forget $C$ from one of these two ontologies or from both. Grau, Parsia, and Sirin [8] propose to use so-called "E-connections" for merging ontologies. In this approach, merging ontologies is defined in terms of link properties. However, it is difficult to find related link properties.

Similar to other approaches to semantic integration, it is a hard issue to determine the set of concepts which should be forgotten if they are not explicitly specified by the user. This issue could be solved by employing heuristics and techniques from machine learning. Exploring this is left for future work.

## References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
2. W. Chen, M. Kifer, and D. Warren. HILOG: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
3. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (*IJCAI 2005*), pages 90–96. Morgan Kaufmann, 2005.
4. T. Eiter and K. Wang. Forgetting and Conflict Resolving in Disjunctive Logic Programming. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (*AAAI 2006*). AAAI Press, 2006.
5. W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence* (*JELIA 2004*), pages 200–212, 2004.
6. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

7. S. Ghilardi, C. Lutz, and F. Wolter. Did I Damage My Ontology? A Case for Conservative Extensions in Description Logics. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 187–197. AAAI Press, 2006.
8. B. Grau, B. Parsia, and E. Sirin. Combining OWL Ontologies using E-Connections. *Journal of Web Semantics*, 4(1), 2005.
9. B. C. Grau, I. Horrocks, O. Kutz, and U. Sattler. Will my Ontologies Fit Together? In *Proceedings of the 2006 International Workshop on Description Logics* (*DL 2006*), 2006.
10. B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and Web Ontologies. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 198–208. AAAI Press, 2006.
11. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proceedings of the Twelfth International World Wide Web Conference* (*WWW 2003*), pages 48–57, 2003.
12. Y. Kalfoglou and M. Schorlemmer. Ontology Mapping: the State of the Art. *The Knowledge Engineering Review*, 18:1–31, 2003.
13. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.
14. J. Lang, P. Liberatore, and P. Marquis. Propositional Independence: Formula-Variable Independence and Forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
15. F. Lin and R. Reiter. Forget it. In *Proceedings of the AAAI Fall Symposium on Relevance*, pages 154–159. New Orleans, 1994.
16. B. Motik, R. Volz, and A. Maedche. Optimizing Query Answering in Description Logics using Disjunctive Deductive Databases. In *Proceedings of the Tenth International Workshop on Knowledge Representation meets Databases* (*KRDB 2003*), 2003. `http://CEUR-WS.org/Vol79/`.
17. N. Noy. Semantic Integration: A Survey of Ontology-Based Approaches. *SIGMOD Record*, 33(4):65–70, 2004.
18. N. Noy and H. Stuckenschmidt. Ontology Alignment: An Annotated Bibliography. In *Semantic Interoperability and Integration*, 2005.
19. K. A. Ross. On Negation in HiLog. *Journal of Logic Programming*, 18(1):27–53, 1994.
20. J. Seidenberg and A. Rector. Web Ontology Segmentation: Analysis, Classification and Use. In *Proceedings of the Fifteenth International World Wide Web Conference* (*WWW 2006*), 2006.
21. T. Swift. Deduction in Ontologies via ASP. In *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR-7*), volume 2923 of *LNCS*, pages 275–288, 2004.
22. K. Wang, A. Sattar, and K. Su. A Theory of Forgetting in Logic Programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (*AAAI 2005*), pages 682–687. AAAI Press, 2005.