Linking Data to Ontologies

Antonella Poggi[1], Domenico Lembo[1], Diego Calvanese[2],
Giuseppe De Giacomo[1], Maurizio Lenzerini[1], and Riccardo Rosati[1]

[1] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy
poggi,degiacomo,lembo,lenzerini,rosati@dis.uniroma1.it

[2] Faculty of Computer Science, Free University of Bozen-Bolzano,
Piazza Domenicani 3, I-39100 Bolzano, Italy
calvanese@inf.unibz.it

Abstract. Many organizations nowadays face the problem of accessing existing data sources by means of flexible mechanisms that are both powerful and efficient. Ontologies are widely considered as a suitable formal tool for sophisticated data access. The ontology expresses the domain of interest of the information system at a high level of abstraction, and the relationship between data at the sources and instances of concepts and roles in the ontology is expressed by means of mappings. In this paper we present a solution to the problem of designing effective systems for ontology-based data access. Our solution is based on three main ingredients. First, we present a new ontology language, based on Description Logics, that is particularly suited to reason with large amounts of instances. The second ingredient is a novel mapping language that is able to deal with the so-called impedance mismatch problem, i.e., the problem arising from the difference between the basic elements managed by the sources, namely data, and the elements managed by the ontology, namely objects. The third ingredient is the query answering method, that combines reasoning at the level of the ontology with specific mechanisms for both taking into account the mappings and efficiently accessing the data at the sources.

1 Introduction

In several areas, such as Enterprise Application Integration, Data Integration [19], and the Semantic Web [13], ontologies are considered as the ideal formal tool to provide a shared conceptualization of the domain of interest. In particular, in many of the above areas, ontologies are advocated for realizing what we can call *ontology-based data access*, that can be explained as follows: we have a set of pre-existing data sources forming the data layer of our information system, and we want to build a service on top of this layer, aiming at presenting a conceptual view of data to the clients of the information system. Specifically, the conceptual view is expressed in terms of an ontology, that will represent the unique access point for the interaction between the clients and the system, and the data sources are independent from the ontology. In other words, our aim is

to link to the ontology a collection of data that exist autonomously, and have not been necessarily structured with the purpose of storing the ontology instances.

Therefore, in ontology-based data access, the ontology describes the domain of interest at a high level of abstraction, so as to abstract away from how data sources are maintained in the data layer of the system itself. It follows that the conceptual view and the data sources are both at different abstraction levels, and expressed in terms of different formalisms. For example, while logical languages are nowadays used to specify the ontology, the data sources are usually expressed in terms of the relational data model.

Taking into account these differences, the specific issues arising from the interaction between the ontology and the data sources can be briefly summarized as follows:

1. Ontologies exhibit to the client a conceptual view of the domain of interest, and allow for expressing at the intensional level complex kinds of semantic conditions over such domain. One of the main challenges in this respect is to single out ontology languages that provide an acceptable compromise between expressive power and computational complexity of reasoning over both the ontology and the underlying sources storing data about the domain.
2. The amount of data stored at the sources can be very large. Therefore, one needs to resort to a technology that is able to efficiently access very large quantities of data. Nowadays, relational database technology is the best (if not the only) one that fulfills such a requirement. Hence, in our context, we are interested in determining how much one can push the expressive power of the formalism used for expressing the conceptual layer, while still maintaining the ability to answer queries by relying on a relational DBMS to access data at the sources.
3. Since we assume that the data sources exist in the information system independently of the conceptual layer, the whole system will be based on specific mechanisms for mapping the data at the sources to the elements of the ontology. So, in ontology-based data access, the mapping is the formal tool by which we determine how to link data to the ontology, i.e., how to reconstruct the semantics of data stored in the sources in terms of the ontology.
4. In general, there is a mismatch between the way in which data is (and can be) represented in a relational database, and the way in which the corresponding information is rendered in an ontology. Specifically, while the database of a data source stores data, instances of concepts in an ontology are objects, each one denoted by an object identifier, not to be confused with a data value. Such a problem is known as *impedance mismatch*. The language used to specify the mappings between the data and the ontology should provide specific mechanisms for addressing the impedance mismatch problem.
5. The main reason to build an ontology-based data access system is to provide high-level services to the clients of the information system. The most important service is query answering. Clients express their queries in terms of the conceptual view (the ontology), and the system should reason about the ontology and the mapping and should translate the request into suitable queries posed to the sources.

Recent research in the area of ontology languages for the Semantic Web has addressed several important aspects regarding the issues mentioned above.

As for issue 1, an effort has been undertaken to understand which language would be best suited for representing ontologies in a setting where an ontology is used for accessing large quantities of data [7, 26, 17]. This work has shown that most of the languages proposed so far are not really suited for this task. Indeed, the most significant fragments of OWL [14][3] that have been proposed by the W3C (namely, OWL-DL and OWL-Lite) are actually coNP-hard in data complexity [10, 7], i.e., when complexity is measured with respect to the size of the data layer only, which is indeed the dominant parameter in this context [31]. This means that, in practice, computations over large amounts of data are prohibitively costly. A way to overcome such a problem is to impose restrictions on the ontology language, so as to guarantee that reasoning remains computationally tractable with respect to data complexity. Possible restrictions that guarantee polynomial reasoning have been studied and proposed in the context of description logics, such as Horn-$\mathcal{SHIQ}$ [17], $\mathcal{EL}^{++}$ [3], and DLP [12]. Among such fragments, of particular interest are those belonging to the *DL-Lite* family [6, 7]. These logics allow for answering complex queries (namely, conjunctive queries, i.e., SQL select-project-join queries, and unions of conjunctive queries) in LogSpace with respect to data complexity. More importantly, after a preprocessing phase which is independent of the data, they allow for delegating query processing to the relational DBMS managing the data layer.

Hence, by adopting a technology based on logics of the *DL-Lite* family, we also aim at a solution to issue 2 above. Specifically, according to [7] there are two maximal languages in the *DL-Lite* family that allow for delegating query processing to a DBMS. The first one, called *DL-Lite*$_{\mathcal{F}}$ in [7], allows for specifying the main modeling features of conceptual models, including cyclic assertions, ISA on concepts, inverses on roles, domain and range of roles, mandatory participation to roles, and functional restrictions on roles. The second one, called *DL-Lite*$_{\mathcal{R}}$, is able to fully capture (the DL fragment of) RDFS, and has in addition the ability of specifying mandatory participation to roles and disjointness between concepts and roles. The language obtained by unrestrictedly merging the features of *DL-Lite*$_{\mathcal{F}}$ and *DL-Lite*$_{\mathcal{R}}$, while quite interesting in general, is not in LogSpace with respect to data complexity anymore [7], and hence looses the most interesting computational feature for ontology-based data access. Hence, to obtain a language whose expressive power goes beyond that of *DL-Lite*$_{\mathcal{F}}$ or *DL-Lite*$_{\mathcal{R}}$ and that is still useful, we need to restrict how the features of both languages are merged and can interact.

Regarding issues 3 and 4, i.e., the impedance mismatch between data items in the data layer and objects at the conceptual level, we observe that such a problem has received only little attention in the Semantic Web community. Some of the issues that need to be addressed when putting into correspondence a relational data source with an ontology, arise also in the context of ontology integration and alignment, which is the topic of several recent research works. These works study

[3] `http://www.w3.org/TR/owl-features/`

formalisms for specifying the correspondences between elements (concepts, relations, individuals) in different ontologies, ranging from simple correspondences between atomic elements, to complex languages allowing for expressing complex mappings. We now briefly discuss the most significant of such proposals found in the literature.

C-OWL and DDLs (Distributed Description Logics) [30] are extensions of OWL and DLs with so-called *bridge rules*, expressing simple forms of semantic relations between concepts, roles, and individuals. At the semantic level, the sets of objects in two ontologies are disjoint, but objects are related to each other by means of *domain relations*, which model simple translation functions between the domains. Reasoning in C-OWL is based on tableaux techniques. MAFRA [23] is a system that allows one to extract mappings from ontologies, and to use them for the transformation of data between ontologies. It does so by providing a so-called *Semantic Bridge Ontology*, whose instantiation provides the ontology mapping, and which can also be used as input for data transformations. The Ontology Mapping Language [29] of the Ontology Management Working Group (OMWG)[4] is an ontology alignment language that is independent of the language in which the two ontologies to be aligned are expressed. The alignment between two ontologies is represented through a set of mapping rules that specify a correspondence between various entities, such as concepts, relations, and instances. Several concept and relation constructors are offered to construct complex expressions to be used in mappings.

While the above proposals deal with the alignment between ontologies, none of them addresses properly the problem of establishing sound mechanisms for linking existing data to the instances of the concepts and the roles in the ontology. This issue is studied in [11, 5], where specific mapping languages are proposed for linking data to ontologies. Such approaches, however, do not deal with the problem of the *impedance mismatch* between objects and values, which needs to be addressed by defining suitable mechanisms for *mapping* the data values to the objects in the ontology, and specifying how object identifiers can be built starting from data values. Instead, such a problem has already been considered in databases, and specifically in the context of declarative approaches to data integration. For example, in [9], a mechanism is proposed for annotating the mappings from the data to a global schema (which plays the role of an ontology). Such annotations, together with specific conversion and matching predicates, specify which attributes should be used to identify objects at the conceptual level, and how data coming from different data sources should be joined. We also mention the work done in deductive object-oriented databases on query languages with invention of objects [15, 16]. Such objects are created starting from values specified in the body of a query, by applying suitable (Skolem) functions.

We argue that the results of the above mentioned papers, although interesting from several points of view, do not provide a clear and comprehensive solution to the problem of designing effective and efficient tools for ontology-based data

[4] `http://www.omwg.org/`

access. The goal of this paper is to present one such solution. Specifically, we present three contributions towards this end:

- We propose a new logic of the *DL-Lite* family. By looking at the interaction between the distinguishing features of *DL-Lite*$_{\mathcal{F}}$ and *DL-Lite*$_{\mathcal{R}}$, we have been able to single out an extension of both logics that is still LogSpace with respect to data complexity, and allows for delegating the "data dependent part" of the query answering process to the relational DBMS managing the data layer. In devising this logic, called *DL-Lite*$_{\mathcal{A}}$, we take seriously the distinction between objects and values (a distinction that is typically blurred in description logics), and introduce, besides concepts and roles, also attributes, which describe properties of concepts represented by values rather than objects.
- We illustrate a specific language for expressing mappings between data at the sources and instances of concepts and roles in the ontology. The mapping language has been designed in such a way to provide a solution to the impedance mismatch problem. Indeed, with respect to previous proposals of mapping languages, the distinguishing feature of our proposal is the possibility to create new object identifiers by making use of values retrieved from the database. We have borrowed this idea from the work mentioned above on query languages with invention of objects [15, 16]. With respect to these works, our approach looks technically simpler, since the mapping mechanism used to create object terms does not allow for recursion. However, we have to deal with the complex constructs presented in the ontology, which significantly complicates matters.
- Our mapping mechanism also deals with the fact that the data sources and the ontology $\mathcal{O}_m$ are based on different semantical assumptions. Indeed, the semantics of data sources follows the so-called "closed world assumption" [28], which intuitively sanctions that every fact that is not explicitly stored in the database is false. On the contrary, the semantics of the ontology is open, in the sense that nothing is assumed about the facts that do not appear explicitely in the ABox.
- We devise a novel query answering method, which is able to fully take into account both the ontology and the mappings from the data layer to the ontology itself. The method extends the ones already presented in [7] for the two sub-logics *DL-Lite*$_{\mathcal{F}}$ and *DL-Lite*$_{\mathcal{R}}$. Similar to these, it works by first *expanding* the query according to the constraints in the ontology. In this case, however, the expanded query is not directly used to compute the result. Rather, the expanded query is the input of a novel step, called *unfolding*, that, taking into account the mappings, translates the expanded query in terms of the relations at the sources. The unfolded query is then evaluated at the sources, and the result is processed so as to conform to the concepts and roles in the ontology. The unfolding step relies on techniques from partial evaluation [21], and the whole query answering method runs in LogSpace in data complexity, i.e., the complexity measured with respect to the size of source data.

The rest of the paper is organized as follows. In Section 2 we present the description logic we deal with, namely *DL-Lite*$_\mathcal{A}$. In Section 3 we present the framework for linking external data sources to an ontology expressed in *DL-Lite*$_\mathcal{A}$. In Section 4 we provide an overview the query answering method, and in Sections 5 and 6 we provide the technical details of such method. Finally, Section 7 concludes the paper.

2 The Description Logic *DL-Lite*$_\mathcal{A}$

Description Logics (DLs) [4] are logics that represent the domain of interest in terms of *concepts*, denoting sets of objects, and *roles*, denoting binary relations between (instances of) concepts. Complex concept and role expressions are constructed starting from a set of atomic concepts and roles by applying suitable constructs. Different DLs allow for different constructs. A DL ontology is constituted by a TBox and an ABox, where the first component specifies general properties of concepts and roles, whereas the second component specifies the instances of concepts and roles.

The study of the trade-off between expressive power and computational complexity of reasoning has been traditionally one of the most important issues in DLs. Recent research has shown that OWL, the W3C Web Ontology Language for the Semantic Web[5], if not restricted, is not suited as a formalism for representing ontologies with large amounts of instance assertions in the ABox [7, 26, 17], since reasoning in such a logic is inherently exponential (coNP-hard) with respect data complexity, i.e., with respect to the size of the ABox.

On the contrary, the *DL-Lite* family [6–8] is a family of DLs specifically tailored to capture basic ontology languages, conceptual data models (e.g., Entity-Relationship [1]), and object-oriented formalisms (e.g., basic UML class diagrams[6]) while keeping the complexity of reasoning low. In particular, ontology satisfiability, instance checking, and answering conjunctive queries in these logics can all be done in LogSpace with respect to data complexity.

In this section, we present a new logic of the *DL-Lite* family, called *DL-Lite*$_\mathcal{A}$. Such a DL is novel with respect to the other DLs of the *DL-Lite* family, in that it takes seriously the distinction between objects and values, and therefore distinguishes:

- concepts from value-domains – while a concept is abstraction for a set of objects, a value-domain, also known as concrete domain [22], denotes a set of (data) values,
- attributes from roles – while a role denotes a binary relation between objects, a (concept) attribute denotes a binary relation between objects and values.

We notice that the distinction between objects and values, although present in OWL, is typically blurred in many DLs. In the following, we first illustrate

[5] `http://www.w3.org/TR/owl-features/`
[6] `http://www.omg.org/uml/`

the mechanisms provided by *DL-Lite*$_\mathcal{A}$ for building expressions, and then we describe how expressions are used to specify ontologies, and which is the form of queries allowed in our logic. Finally, we conclude the section by describing relevant reasoning tasks over *DL-Lite*$_\mathcal{A}$ ontologies.

2.1 *DL-Lite*$_\mathcal{A}$ expressions

Like in any other logics, *DL-Lite*$_\mathcal{A}$ expressions are built over an alphabet. In our case, the alphabet comprises symbols for atomic concepts, value-domains, atomic roles, atomic attributes, and constants.

The value-domains that we consider in *DL-Lite*$_\mathcal{A}$ are those corresponding to the data types adopted by the Resource Description Framework (RDF)[7]. Intuitively, these types represent sets of values that are pairwise disjoint. In the following, we denote such value-domains by $T_1, \ldots, T_n$.

Furthermore, we denote with Γ the alphabet for constants, which we assume partitioned into two sets, namely, Γ_V (the set of constant symbols for values), and Γ_O (the set of constant symbols for objects). In turn, Γ_V is partitioned into n sets $\Gamma_{V_1}, \ldots, \Gamma_{V_n}$, where each Γ_{V_i} is the set of constants for the values in the value-domain T_i.

In providing the specification of our logics, we use the following notation:

- A denotes an *atomic concept*, B a *basic concept*, C a *general concept*, and $\top_C$ denotes the *universal concept*. An atomic concept is a concept denoted by a name. Basic and general concepts are concept expressions whose syntax is given at point 1 below.
- E denotes a basic value-domain, i.e., the range of an attribute, F a *value-domain expression*, and $\top_D$ the *universal value-domain*. The syntax of value-domain expressions is given at point 2 below.
- P denotes an *atomic role*, Q a *basic role*, and R a *general role*. An atomic role is simply a role denoted by a name. Basic and general roles are role expressions whose syntax is given at point 3 below.
- U_C denotes an *atomic attribute* (or simply attribute), and V_C a *general attribute*. An atomic attribute is an attribute denoted by a name, whereas a general attribute is a concept expression whose syntax is given at point 4 below.

Given an attribute U_C, we call the *domain* of U_C, denoted by $\delta(U_C)$, the set of objects that U_C relates to values, and we call *range* of U_C, denoted by $\rho(U_C)$, the set of values that U_C relates to objects. Note that the domain $\delta(U_C)$ of an attribute U_C is a concept, whereas the range $\rho(U_C)$ of U_C is a value-domain.

We are now ready to define *DL-Lite*$_\mathcal{A}$ expressions.

1. Concept expressions:

$$\begin{array}{rcl} B & ::= & A \mid \exists Q \mid \delta(U_C) \\ C & ::= & \top_C \mid B \mid \neg B \mid \exists Q.C \end{array}$$

[7] `http://www.w3.org/RDF/`

2. Value-domain expressions:

$$\begin{array}{rcl} E & ::= & \rho(U_C) \\ F & ::= & \top_D \mid T_1 \mid \cdots \mid T_n \end{array}$$

3. Role expressions:

$$\begin{array}{rcl} Q & ::= & P \mid P^- \\ R & ::= & Q \mid \neg Q \end{array}$$

4. Attribute expressions:

$$V_C ::= U_C \mid \neg U_C$$

The meaning of every *DL-Lite*$_\mathcal{A}$ expression is sanctioned by the semantics. Following the classical approach in DLs, the semantics of *DL-Lite*$_\mathcal{A}$ is given in terms of first-order logic interpretations. All such intepretations agree on the semantics assigned to each value-domain T_i and to each constant in Γ_V. In particular, each T_i is interpreted as the set $val(T_i)$ of values of the corresponding RDF data type, and each $c_i \in \Gamma_V$ is interpreted as one specific value, denoted $val(C_i)$, in $val(T_i)$. Note that, for $i \neq j$, it holds that $val(T_i) \cap val(T_j) = \emptyset$.

Based on the above observations, we can now define the notion of interpretation in *DL-Lite*$_\mathcal{A}$. An *interpretation* is a pair $I = (\Delta^I, \cdot^I)$, where

- Δ^I is the interpretation domain, that is the disjoint union of two non-empty sets: Δ_O^I, called the *domain of objects*, and Δ_V^I, called the *domain of values*. In turn, Δ_V^I is the union of $val(T_1), \ldots, val(T_n)$.
- $\cdot^I$ is the *interpretation function*, i.e., a function that assigns an element of Δ^I to each constant in Γ, a subset of Δ^I to each concept and value-domain, and a subset of $\Delta^I \times \Delta^I$ to each role and attribute, in such a way that
  - for each $a \in \Gamma_V$, $a^I = val(a)$,
  - for each $a \in \Gamma_O$, $a^I \in \Delta_O^I$,
  - for each $a, b \in \Gamma$, $a \neq b$ implies $a^I \neq b^I$,
  - for each T_i, $T_i^I = val(T_i)$,
  - the following conditions are satisfied:

$$\begin{array}{ll} \top_C^I = \Delta_O^I & (\rho(U_C))^I = \{\ v \mid \exists o.\, (o, v) \in U_C^I\ \} \\ \top_D^I = \Delta_V^I & (\delta(U_C))^I = \{\ o \mid \exists o.\, (o, v) \in U_C^I\ \} \\ A^I \subseteq \Delta_O^I & (P^-)^I = \{\ (o, o') \mid (o', o) \in P^I\ \} \\ P^I \subseteq \Delta_O^I \times \Delta_O^I & (\exists Q)^I = \{\ o \mid \exists o'.\, (o, o') \in Q^I\ \} \\ U_C^I \subseteq \Delta_O^I \times \Delta_V^I & (\neg Q)^I = (\Delta_O^I \times \Delta_O^I) \setminus Q^I \\ (\neg U_C)^I = (\Delta_O^I \times \Delta_V^I) \setminus U_C^I & (\neg B)^I = \Delta_O^I \setminus B^I \end{array}$$

Note that the above definition implies that different constants are interpreted differently in the domain, i.e., *DL-Lite*$_\mathcal{A}$ adopts the so-called unique name assumption.

2.2 *DL-Lite*$_\mathcal{A}$ ontologies

As usual when expressing ontologies in DLs, a *DL-Lite*$_\mathcal{A}$ *ontology* $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ represents the domain of discourse in terms of two components: the TBox $\mathcal{T}$,

representing the intensional knowledge, and the ABox $\mathcal{A}$, representing the extensional knowledge. *DL-Lite*$_\mathcal{A}$ TBoxes and ABoxes are defined as follows. *DL-Lite*$_\mathcal{A}$ *intensional assertions* are assertions of the form:

$$\begin{array}{ccc} B & \sqsubseteq & C \quad \textit{(concept inclusion assertion)} \\ Q & \sqsubseteq & R \quad \textit{(role inclusion assertion)} \\ E & \sqsubseteq & F \quad \textit{(value-domain inclusion assertion)} \\ U_C & \sqsubseteq & V_C \quad \textit{(attribute inclusion assertion)} \\ (\mathsf{funct}\ Q) & & \textit{(role functionality assertion)} \\ (\mathsf{funct}\ U_C) & & \textit{(attribute functionality assertion)} \end{array}$$

A concept (respectively, value-domain, role, and attribute) inclusion assertion expresses that a basic concept B (respectively, basic value-domain E, basic role Q, and atomic attribute U_C) is subsumed by a general concept C (respectively, value-domain F, role R, attribute V_C). A role functionality assertion expresses the (global) functionality of a role. In the case where $Q = P$, the functionality constraint is imposed on an atomic role, while in the case where $Q = INVP$, it is imposed on the inverse of an atomic role. Analogously, an attribute functionality assertion expresses the (global) functionality of an atomic attribute. Concept (respectively, value-domain, and role) inclusions of the form $B_1 \sqsubseteq \neg B_2$ (respectively, $E_1 \sqsubseteq \neg E_2$, $Q_1 \sqsubseteq \neg Q_2$) are called *negative inclusion assertions*.

Then, *DL-Lite*$_\mathcal{A}$ TBoxes are finite sets of *DL-Lite*$_\mathcal{A}$ intensional assertions where suitable limitations in the combination of such assertions are imposed. To precisely describe such limitations, we first introduce some preliminary notions. An atomic attribute U_C (respectively, a basic role Q) is called an *identifying property in a TBox* $\mathcal{T}$, if $\mathcal{T}$ contains a functionality assertion $(\mathsf{funct}\ U_C)$ (respectively, $(\mathsf{funct}\ Q)$). Let X be an atomic attribute or a basic role. We say that X *appears positively* (respectively, *negatively*) in the right-hand side of an inclusion assertion α if α has the form $Y \sqsubseteq X$ (respectively, $Y \sqsubseteq \neg X$). Also, an atomic attribute or a basic role is called *primitive in a TBox* $\mathcal{T}$, if it does not appear positively in the right-hand side of an inclusion assertion of $\mathcal{T}$, and it does not appear in an expression of the form $\exists Q.C$ in $\mathcal{T}$. Then,

> *a DL-Lite*$_\mathcal{A}$ *TBox is a finite set* $\mathcal{T}$ *of DL-Lite*$_\mathcal{A}$ *intensional assertions satisfying the condition that every identifying property in* $\mathcal{T}$ *is primitive in* $\mathcal{T}$*.*

Roughly speaking, in a *DL-Lite*$_\mathcal{A}$ TBox, *identifying properties cannot be specialized*, i.e., they cannot appear positively in the right-hand side of inclusion assertions.

We now specify the semantics of a TBox $\mathcal{T}$, again in terms of interpretations. An interpretation I satisfies

- a concept (respectively, value-domain, role, attribute) inclusion assertion $B \sqsubseteq C$ (respectively, $E \sqsubseteq F$, $Q \sqsubseteq R$, $U_C \sqsubseteq V_C$), if

$$B^I \subseteq C^I \qquad \text{(respectively, } E^I \subseteq F^I, Q^I \subseteq R^I, U_C^I \subseteq V_C^I\text{)}$$

– a role functionality assertion (funct Q), if for each $o_1, o_2, o_3 \in \Delta_O^I$

$$(o_1, o_2) \in Q^I \text{ and } (o_1, o_3) \in Q^I \text{ implies } o_2 = o_3$$

– an attribute functionality assertion (funct U_C), if for each $o \in \Delta_O^I$ and $v_1, v_2 \in \Delta_V^I$

$$(o, v_1) \in U_C^I \text{ and } (o, v_2) \in U_C^I \text{ implies } v_1 = v_2.$$

I is a *model* of a *DL-Lite*$_\mathcal{A}$ TBox $\mathcal{T}$, or, equivalently, I *satisfies* $\mathcal{T}$, written $I \models \mathcal{T}$, if and only if I satisfies all intensional assertions in $\mathcal{T}$.

We next illustrate an example of a *DL-Lite*$_\mathcal{A}$ TBox. In all the examples of this paper, we write concept names in *lowercase*, role names in *UPPERCASE*, attribute names in sans serif font, and domain names in `typewriter` font.

Example 1. Let $\mathcal{T}$ be the TBox containing the following assertions:

$\mathit{tempEmp} \sqsubseteq \mathit{employee}$	(1)	$\mathit{project} \sqsubseteq \delta(\mathsf{ProjName})$	(9)
$\mathit{manager} \sqsubseteq \mathit{employee}$	(2)	(funct ProjName)	(10)
$\mathit{employee} \sqsubseteq \mathit{person}$	(3)	$\rho(\mathsf{ProjName}) \sqsubseteq$ `xsd:string`	(11)
$\mathit{employee} \sqsubseteq \exists \mathit{WORKS\text{-}FOR}$	(4)	$\mathit{tempEmp} \sqsubseteq \delta(\mathsf{until})$	(12)
$\exists \mathit{WORKS\text{-}FOR}^- \sqsubseteq \mathit{project}$	(5)	$\delta(\mathsf{until}) \sqsubseteq \exists \mathit{WORKS\text{-}FOR}$	(13)
$\mathit{person} \sqsubseteq \delta(\mathsf{PersName})$	(6)	(funct until)	(14)
(funct PersName)	(7)	$\rho(\mathsf{until}) \sqsubseteq$ `xsd:date`	(15)
$\rho(\mathsf{PersName}) \sqsubseteq$ `xsd:string`	(8)	$\mathit{manager} \sqsubseteq \neg\delta(\mathsf{until})$	(16)

The above TBox $\mathcal{T}$ models information about employees and projects they work for. Specifically, the assertions in $\mathcal{T}$ state the following. Managers and temporary employees are two kinds of employees (2, 1), and employees are persons (3). Each employee works for at least one project (4, 5), whereas each person and each project has a unique name (6, 7, 9, 10). Both person names and project names are strings (8, 11), whereas the attribute until associates objects with dates (14, 15). In particular, any temporary employee has an associated date (which indicates the expiration date of her/his contract) (12), and everyone having a value for attribute until participates in the role *WORKS-FOR* (13). Finally, $\mathcal{T}$ specifies that a manager does not have any value for the attribute until (16), meaning that a manager has a permanent position, Note that this implies that no employee is simultaneously a temporary employee and a manager. □

We now specify the form of *DL-Lite*$_\mathcal{A}$ ABoxes. A *DL-Lite*$_\mathcal{A}$ ABox is a finite set of assertions, called *membership assertions*, of the form:

$$A(a), \quad P(a, b), \quad U_C(a, b)$$

where a and b are constants in the alphabet Γ.

As for the semantics of a *DL-Lite*$_\mathcal{A}$ ABox $\mathcal{A}$, we now specify when an interpretation $I = (\Delta^I, \cdot^I)$ satisfies a membership assertion α in $\mathcal{A}$, written $I \models \alpha$. I satisfies:

- $A(a)$ if $a^I \in A^I$;
- $P(a, b)$ if $(a^I, b^I) \in P^I$;
- $U_C(a, b)$ if $(a^I, b^I) \in U_C^I$.

I is *model of* $\mathcal{A}$, or, equivalently, I *satisfies* $\mathcal{A}$, written $I \models \mathcal{A}$, if I satisfies all the membership assertions in $\mathcal{A}$.

We next illustrate an example of a *DL-Lite*$_\mathcal{A}$ ABox. In the example, we use the **bold face** font for constants in Γ_O, and the *slanted* font for constants in Γ_V.

Example 2. Consider the following ABox $\mathcal{A}$:

$$\textit{tempEmp}(\textbf{Palm}) \tag{17}$$
$$\mathsf{until}(\textbf{Palm}, \textit{25-09-05}) \tag{18}$$
$$\mathsf{ProjName}(\textbf{DIS-1212}, \textit{QuOnto}) \tag{19}$$
$$\textit{manager}(\textbf{White}) \tag{20}$$
$$\textit{WORKS-FOR}(\textbf{White}, \textbf{FP6-7603}) \tag{21}$$
$$\mathsf{ProjName}(\textbf{FP6-7603}, \textit{Tones}) \tag{22}$$

The ABox assertions in $\mathcal{A}$ state that the object (identified by the constant) **Palm** denotes a temporary employee who works until the date *25-09-05*. Moreover, **DIS-1212** and **FP6-7603** are projects whose names are respectively *QuOnto* and *Tones*. Finally, the object **White** is a manager. □

Now that we have introduced *DL-Lite*$_\mathcal{A}$ TBoxes and ABoxes, we are able to define the semantics of a *DL-Lite*$_\mathcal{A}$ ontology, which is given in terms of interpretations which satisfy both the TBox and the ABox of the ontology. More formally, an interpretation $I = (\Delta^I, \cdot^I)$ is *model of a DL-Lite*$_\mathcal{A}$ *ontology* $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, or, equivalently, I *satisfies* $\mathcal{O}$, written $I \models \mathcal{O}$, if both $I \models \mathcal{T}$ and $I \models \mathcal{A}$. We say that $\mathcal{O}$ is *satisfiable* if it has at least one model.

Example 3. Let $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ be the *DL-Lite*$_\mathcal{A}$ ontology whose TBox $\mathcal{T}$ is the one of Example 1, and whose ABox $\mathcal{A}$ is the one of Example 2. The first observation is that $\mathcal{O}$ is satisfiable. Furthermore, it is easy to see that every model $I = (\Delta^I, \cdot^I)$ of $\mathcal{A}$ satisfies the following conditions:

$$\textbf{Palm}^I \in \textit{tempEmp}^I$$
$$(\textbf{Palm}^I, \textit{25-09-05}^I) \in \mathsf{until}^I$$
$$(\textbf{DIS-1212}^I, \textit{QuOnto}^I) \in \mathsf{ProjName}^I$$
$$\textbf{White}^I \in \textit{manager}^I$$
$$(\textbf{White}^I, \textbf{FP6-7603}^I) \in \textit{WORKS-FOR}^I$$
$$(\textbf{FP6-7603}^I, \textit{Tones}^I) \in \mathsf{ProjName}^I.$$

Furthermore, the following are necessary conditions for I to be a model of the TBox $\mathcal{T}$ (we indicate in parenthesis the reference to the relevant axiom of $\mathcal{T}$):

$\mathbf{Palm}^I \in \mathit{employee}^I$, to satisfy inclusion assertion (1)
$\mathbf{White}^I \in \mathit{employee}^I$, to satisfy inclusion assertion (2)
$\mathbf{Palm}^I \in \mathit{person}^I$, to satisfy inclusion assertion (3)
$\mathbf{White}^I \in \mathit{person}^I$, to satisfy inclusion assertion (3)
$\mathbf{Palm}^I \in \exists \mathit{WORKS\text{-}FOR}^I$, to satisfy inclusion assertion (4)
$\mathbf{FP6\text{-}7603}^I \in \mathit{project}^I$, to satisfy inclusion assertion (5)
$\mathbf{Palm}^I \in (\delta(\mathsf{PersName}))^I$, to satisfy inclusion assertion (6)
$\mathbf{White}^I \in (\delta(\mathsf{PersName}))^I$, to satisfy inclusion assertion (6)

Notice that, in order for an interpretation I to satisfy the condition specified in the fifth row above, there must be an object $o \in \Delta_O^I$ such that $(\mathbf{Palm}^I, o) \in \mathit{WORKS\text{-}FOR}^I$. According to the inclusion assertion (5), such an object o must also belong to $\mathit{project}^I$ (indeed, in our ontology, every employee works for at least one project). Similarly, the last two rows above derive from the property that every person must have a name (inclusion (6)).

We note that, besides satisfying the conditions discussed above, an interpretation I' may also add other elements to the interpretation of concepts, attributes, and roles specified by I. For instance, the interpretation I' which adds to I the tuple

$$(\mathbf{White}^{I'}, \mathbf{DIS\text{-}1212}^{I'}) \in \mathit{WORKS\text{-}FOR}^{I'}$$

is still a model of the ontology.

Note, finally, that there exists no model of $\mathcal{O}$ such that **White** is interpreted as a temporary employee, since, according to (20), **White** is a manager and, as observed in Example 1, the sets of managers and temporary employees are disjoint. □

The above example clearly shows the difference between a database and an ontology. From a database point of view the ontology $\mathcal{O}$ discussed in the example might seem incorrect: for example, while the TBOx $\mathcal{T}$ sanctions that every person has a name, there is no explicit name for **White** (who is a person, because he has been asserted to be a manager, and every manager is a person) in the ABox $\mathcal{A}$. However, the ontology is not incorrect: the axiom stating that every person has a name simply specifies that in every model of $\mathcal{O}$ there will be a name for **White**, even if such a name is not known.

2.3 Queries over *DL-Lite*$_\mathcal{A}$ ontologies

We are interested in expressing queries over ontologies expressed in *DL-Lite*$_\mathcal{A}$, and similarly to the case of relational databases, the basic query class that we consider is the class of conjunctive queries.

A *conjunctive query* (CQ) q over a *DL-Lite*$_\mathcal{A}$ ontology is an expression of the form

$$q(\boldsymbol{x}) \leftarrow \mathit{conj}(\boldsymbol{x}, \boldsymbol{y})$$

where $\boldsymbol{x}$ is a tuple of distinct variables, the so-called *distinguished variables*, $\boldsymbol{y}$ is a tuple of distinct existentially quantified variables (not occurring in $\boldsymbol{x}$), called the *non-distinguished* variables, and $conj(\boldsymbol{x}, \boldsymbol{y})$ is a *conjunction* of atoms of the form $A(x)$, $P(x, y)$, $D(x)$, $U_C(x, y)$, $x = y$, where:

- A, P, D, and U_C are respectively an atomic concept, an atomic role, an atomic value-domain, and an atomic attribute in $\mathcal{O}$,
- x, y are either variables in $\boldsymbol{x}$ or in $\boldsymbol{y}$, or constants in Γ.

We say that $q(\boldsymbol{x})$ is the *head* of the query whereas $conj(\boldsymbol{x}, \boldsymbol{y})$ is the *body*. Moreover, the *arity* of q is the arity of $\boldsymbol{x}$.

We will also refer to the notion of *conjunctive query with inequalities* (CQI), that is simply a conjunctive query in which atoms of the $x \neq y$ (called inequalities) may appear. Finally, a *union of conjunctive queries* (UCQ) is a query of the form:

$$Q(\boldsymbol{x}) \leftarrow conj_1(\boldsymbol{x}, \boldsymbol{y_i}) \cup \cdots \cup conj_n(\boldsymbol{x}, \boldsymbol{y_n}).$$

Unions of conjunctive queries with inequalities are obvious extensions of unions of conjunctive queries. In the following, we use the Datalog notation for unions of conjunctive queries. In this notation a union of conjuctive queries is written in the form

$$\begin{aligned} &Q(\boldsymbol{x}) \leftarrow conj_1(\boldsymbol{x}, \boldsymbol{y_1}) \\ &\cdots\cdots\cdots \\ &Q(\boldsymbol{x}) \leftarrow conj_n(\boldsymbol{x}, \boldsymbol{y_n}) \end{aligned}$$

Given an interpretation $I = (\Delta^I, \cdot^I)$, the query $Q(\boldsymbol{x}) \leftarrow \varphi(\boldsymbol{x}, \boldsymbol{y})$ (either a conjunctive query or a union of conjunctive queries) is interpreted in I as the set of tuples $\boldsymbol{o_x} \in \Delta^I \times \cdots \times \Delta^I$ such that there exists $\boldsymbol{o_y} \in \Delta^I \times \cdots \times \Delta^I$ such that if we assign to the tuple of variables $(\boldsymbol{x}, \boldsymbol{y})$ the tuple $(\boldsymbol{o_x}, \boldsymbol{o_y})$, then the formula φ is true in I [1].

Example 4. Let $\mathcal{O}$ be the ontology introduced in Example 3. Consider the following query asking for all employees:

$$q_1(x) \leftarrow employee(x).$$

If I is the model described in Example 3, we have that:

$$q_1^I = \{(\mathbf{White}^I), (\mathbf{Palm}^I)\}.$$

Note that we would obtain an analogous result by considering the model I' introduced in Example 3. Suppose now that we ask for project workers, together with the name of the project s/he works in:

$$q_2(x, y) \leftarrow WORKS\text{-}FOR(x, z), \mathsf{ProjName}(z, y).$$

Then we have the following (we assume that, according to I, p is the project for which $\mathbf{Palm}^I$ works):

- $q_2^I = \{(\mathbf{White}^I, Tones^I), (\mathbf{Palm}^I, p)\}$;

– $q_2^{I'} = \{(\mathbf{White}^{I'}, \mathit{Tones}^{I'}), (\mathbf{White}^{I'}, \mathit{QuOnto}^{I'})\}$. □

Let us now describe what it means to answer a query over a *DL-Lite*$_\mathcal{A}$ ontology. Let $\mathcal{O}$ be a *DL-Lite*$_\mathcal{A}$ ontology, Q a UCQ over $\mathcal{O}$, and $\boldsymbol{t}$ a tuple of elements of Γ. We say that $\boldsymbol{t}$ is a *certain answer* to q over $\mathcal{O}$, written $\boldsymbol{t} \in \mathit{ans}(Q, \mathcal{O})$, if for every model I of $\mathcal{O}$, we have that $\boldsymbol{t}^I \in Q^I$. Answering a query Q posed to an ontology $\mathcal{O}$ means exactly to compute the certain answers.

Example 5. Consider again the ontology introduced in Example 3, and queries q_1, q_2 of Example 4. One can easily verify that the set of certain answers to q_1 is $\{\mathbf{White}, \mathbf{Palm}\}$, whereas the set of certain answers to q_2 is $\{(\mathbf{White}, \mathit{QuOnto})\}$.

2.4 Reasoning over *DL-Lite*$_\mathcal{A}$ ontologies

Our logic *DL-Lite*$_\mathcal{A}$ is equipped with traditional DL reasoning services, such as concept and role subsumption, ontology satisfiability and instance checking. Notably, it can be shown (cf. [8]), that all these services can be reduced to satisfiability and query answering. In the following, we therefore briefly discuss satisfiability and query answering for *DL-Lite*$_\mathcal{A}$ ontologies, and present some important properties of such services. The technical results mentioned in this subsection are easy extensions of analogous results presented in [8, 6, 27].

Before discussing the main properties of our reasoning method, we observe that we assume that the ABox of a *DL-Lite*$_\mathcal{A}$ ontology is represented by a relational database. More precisely, if $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is a *DL-Lite*$_\mathcal{A}$ ontology, then we represent $\mathcal{A}$ in terms of the relational database $\mathit{db}(\mathcal{A})$, defined as follows:

– $\mathit{db}(\mathcal{A})$ contains one unary relation T_A for every atomic concept A appearing in $\mathcal{T}$. Such relation has the tuple t in $\mathit{db}(\mathcal{A})$ if and only if the assertion $A(t)$ is in $\mathcal{A}$.
– $\mathit{db}(\mathcal{A})$ contains one binary relation T_P for every atomic role P appearing in $\mathcal{T}$. Such relation has the tuple t in $\mathit{db}(\mathcal{A})$ if and only if the assertion $P(t)$ is in $\mathcal{A}$.
– $\mathit{db}(\mathcal{A})$ contains one binary relation T_U for every atomic attribute U appearing in $\mathcal{T}$. Such relation has the tuple t in $\mathit{db}(\mathcal{A})$ if and only if the assertion $U(t)$ is in $\mathcal{A}$.

One notable property of *DL-Lite*$_\mathcal{A}$ is that, by virtue of the careful definition of the expressive power of the logic, reasoning over the ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ can be reduced to answering suitable queries over $\mathit{db}(\mathcal{A})$.

As for satisfiability, i.e., the problem of checking whether $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable, it can be shown [8, 27] that such a reasoning task can be reduced to the task of evaluating a suitable query, called $\mathsf{Violates}(\mathcal{T})$. Intuitively, $\mathsf{Violates}(\mathcal{T})$ is a first-order query that asks for all constants in $\mathcal{A}$ violating either:

– explicit constraints corresponding to the functionality and disjointness assertions in $\mathcal{T}$, or

– implicit constraints, following from the semantics of $\mathcal{T}$, namely constraints imposing that every concept is disjoint from every domain, and that, for every pair T_i, T_j of *rdfDataType*, T_i and T_j are disjoint.

We denote with $\mathsf{ViolatesDB}(\mathcal{T})$ the function that transforms the query $\mathsf{Violates}(\mathcal{T})$ by changing every predicate X in $\mathsf{Violates}(\mathcal{T})$ into T_X. Therefore, the query $\mathsf{ViolatesDB}(\mathcal{T})$ is equivalent to $\mathsf{Violates}(\mathcal{T})$, but is expressed over $db(\mathcal{A})$. Also, it is immediate to verify that $\mathsf{ViolatesDB}(\mathcal{T})$ can be expressed in SQL.

The correctness of this reduction is sanctioned by the results of [8, 6, 27], summarized here by the following theorem.

Theorem 1. *The DL-Lite$_\mathcal{A}$ ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable if and only if the result of evaluating $\mathsf{Violates}(\mathcal{T})$ over O_m is the empty set, if and only if the result of evaluating $\mathsf{ViolatesDB}(\mathcal{T})$ over $db(\mathcal{A})$ is the empty set.*

Example 6. Consider the ontology introduced in Example 3. Then, $\mathsf{Violates}(\mathcal{T})$ is a union of conjunctive queries including the following disjuncts (corresponding to explicit constraints):

$$
\begin{aligned}
&Q^s(x) \leftarrow \mathit{manager}(x), \mathsf{until}(x, y)\\
&Q^s(x) \leftarrow \mathsf{PersName}(x, y_1), \mathsf{PersName}(x, y_2), y_1 \neq y_2\\
&Q^s(x) \leftarrow \mathsf{ProjName}(x, y_1), \mathsf{ProjName}(x, y_2), y_1 \neq y_2\\
&Q^s(x) \leftarrow \mathsf{until}(x, y_1), \mathsf{until}(x, y_2), y_1 \neq y_2
\end{aligned}
$$

□

As for query answering, it can be shown that computing the certain answers of a query with respect to a satisfiable *DL-Lite$_\mathcal{A}$* ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ can be reduced, through a process called *perfect reformulation*, to the evaluation over $db(\mathcal{A})$ of a suitable union of conjunctive queries. The crucial task of perfect reformulation is carried out by the function PerfectRef. Informally, PerfectRef takes as input a UCQ Q over $\mathcal{O}_m$ and the TBox $\mathcal{T}$, and reformulates Q into a new query Q', which is still a UCQ and has the following property: the answers to Q' with respect to $\langle \emptyset, \mathcal{M}, \mathcal{DB} \rangle$ coincide with the certain asnwers to Q with respect to $\langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$. Thus, all the knowledge represented by the TBox $\mathcal{T}$ that is relevant for computing the certain answers of the query Q is compiled into $\mathsf{PerfectRef}(Q, \mathcal{T})$.

We denote with $\mathsf{PerfectRefDB}(Q, \mathcal{T})$ the function that transforms the query $\mathsf{PerfectRef}(Q, \mathcal{T})$ by changing every predicate X in $\mathsf{PerfectRef}(Q, \mathcal{T})$ into T_X. Therefore, the query $\mathsf{PerfectRefDB}(Q, \mathcal{T})$ is equivalent to $\mathsf{PerfectRef}(Q, \mathcal{T})$, but is expressed over $db(\mathcal{A})$. Also, it is immediate to verify that $\mathsf{PerfectRefDB}(Q, \mathcal{T})$ can be expressed in SQL.

From the results of [8, 6, 27], we have the following:

Theorem 2. *If $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is a satisfiable DL-Lite$_\mathcal{A}$ ontology, and Q is a union of conjunctive queries over $\mathcal{O}$, then $\boldsymbol{t} \in \mathit{ans}(Q, \mathcal{O})$ if and only if $\boldsymbol{t} \in \mathit{ans}(\mathsf{PerfectRef}(Q, \mathcal{T}), \langle \emptyset, \mathcal{A} \rangle)$, if and only if $\boldsymbol{t}$ is in the result of evaluating $\mathsf{PerfectRefDB}(Q, \mathcal{T})$ over $db(\mathcal{A})$.*

Example 7. Consider again the ontology $\mathcal{O}$ of Example 3 and the query q asking for all workers, i.e., those objects which participate to the *WORKS-FOR* role:

$$q(x) \leftarrow \mathit{WORKS\text{-}FOR}(x, y).$$

It can be shown that $\mathsf{PerfectRef}(q, \mathcal{T})$ is the following query Q^p (that is a UCQ):

$$\begin{aligned} Q^p(x) &\leftarrow \mathit{WORKS\text{-}FOR}(x, y) \\ Q^p(x) &\leftarrow \mathsf{until}(x, y) \\ Q^p(x) &\leftarrow \mathit{tempEmp}(x) \\ Q^p(x) &\leftarrow \mathit{employee}(x) \\ Q^p(x) &\leftarrow \mathit{manager}(x). \end{aligned}$$

By virtue of the above theorem, the result of evaluating Q^p over $db(\mathcal{A})$ coincides with the set of certain answers to q over $\mathcal{O}$. Roughly speaking, in order to return all workers, Q^p looks in those concepts, relations, and attributes, whose extensions in $db(\mathcal{A})$ provide objects that are workers, according to the knowledge specified by $\mathcal{T}$. In our case, the answer to the query is $\{\mathbf{White}, \mathbf{Palm}\}$. □

We finally point out that, from the properties discussed above, namely that both ontology satisfiability and query answering are reducible to first-order query evaluation over a suitable relational database, it follows that, after the reformulation process, the task of computing the certain answers to a query can be delegated to a standard relational DBMS [7]. In turn, this implies that all reasoning tasks in *DL-Lite*$_\mathcal{A}$ can be done in LogSpace with respect to data complexity [8, 27].

3 Linking relational data to *DL-Lite*$_\mathcal{A}$ ontologies

The discussion presented in the previous section on *DL-Lite*$_\mathcal{A}$ ontologies assumed a relational representation for the ABox assertions. This is a reasonable assumption only in those cases where the ontology is managed by an ad-hoc system, and is built from scratch for the specific application.

We argue that this is not a typical scenario in current applications (e.g., in Enterprise Application Integration). As we said in the introduction, we believe that one of the most interesting real-world usages of ontologies is what we call "ontology-based data access". Ontology-based data access is the problem of accessing a set of existing data sources by means of a conceptual representation expressed in terms of an ontology. In such a scenario, the TBox of the ontology provides a shared, uniform, abstract view of the intensional level of the application domain, whereas the information about the extensional level (the instances of the ontology) reside in the data sources that are developed independently of the conceptual layer, and are managed by traditional technologies (such as the relational database technology). In other words, the ABox of the ontology does not exist as an independent syntactic object. Rather, the instances of concepts and roles in the ontology are simply an abstract and virtual representation of

some real data stored in existing data sources. Therefore, the problem arises of establishing sound mechanisms for linking existing data to the instances of the concepts and the roles in the ontology.

In this section we present the basic idea for our solution to this problem, by presenting a mapping mechanism that enables a designer to link existing data sources to an ontology expressed in *DL-Lite*$_\mathcal{A}$, and by illustrating a formal framework capturing the notion of *DL-Lite*$_\mathcal{A}$ ontology with mappings. In the following,we assume that the data sources are expressed in terms of the relational data model. In other words, all the technical development presented in the rest of this section assumes that the set of sources to be linked to the ontology is one relational database. Note that this is a realistic assumption, since many data federation tools are now available that are able to wrap a set of heterogeneous sources and present them as a single relational database.

Before delving into the details of the method, a preliminary discussion on the notorious *impedance mismatch problem* between values (data) and objects is in order [24]. When mapping relational data sources to ontologies, one should take into account that sources store values, whereas instances of concepts are objects, where each object should be denoted by an ad hoc identifier (e.g., a constant in logic), not to be confused with any data item. For example, if a data source stores data about persons, it is likely that values for social security numbers, names, etc. will appear in the sources. However, at the conceptual level, the ontology will represent persons in terms of a concept, and instances of such concepts will be denoted by object constants.

One could argue that data sources might, in some cases, store directly object identifiers. However, in order to use such object identifiers at the conceptual level, one should make sure that such identifiers have been chosen on the basis of an "agreement" among the sources on the form used to represent objects. This is something occurring very rarely in practice. For all the above reasons, in *DL-Lite*$_\mathcal{A}$, we take a radical approach. To face the impedance mismatch problem, and to tackle the possible lack of an a-priori agreement on identification mechanisms at the sources, we keep data values appearing in the sources separate from object identifiers at the conceptual level. In particular, we consider object identifiers formed by (logic) terms built out from data values stored at the sources. The way by which these terms will be defined starting from the data at the sources will be specified through suitable mapping assertions, to be described later in this section. Note that this idea traces back to the work done in deductive object-oriented databases [15].

To realize this idea from a technical point of view, we specialize the alphabets of object constants in a particular way, that we now describe in detail.

We remind the reader that Γ_V is the alphabet of value constants in *DL-Lite*$_\mathcal{A}$. We assume that data appearing at the sources are denoted by constants in Γ_V[8], and we introduce a new alphabet Λ of function symbols in *DL-Lite*$_\mathcal{A}$, where each

[8] We could also introduce suitable conversion functions in order to translate values stored at the sources into value constants in Γ_V, but, for the sake of simplicity, we do not deal with this aspect here.

function symbol has an associated arity, specifying the number of arguments it accepts. On the basis of Γ_V and Λ, we inductively define the set $\tau(\Lambda, \Gamma_V)$ of all *terms* of the form $f(d_1, \ldots, d_n)$ such that

- $f \in \Lambda$,
- the arity of f is $n > 0$, and
- $d_1, \ldots, d_n \in \Gamma_V$.

We finally sanction that the set Γ_O of symbols used in *DL-Lite*$_\mathcal{A}$ for denoting objects actually coincides with $\tau(\Lambda, \Gamma_V)$. In other words, we use the terms built out of Γ_V using the function symbols in Λ for denoting the instances of concepts in *DL-Lite*$_\mathcal{A}$ ontologies.

All the notions defined for our logics remain unchanged. In particular, an interpretation $I = (\Delta^I, \cdot^I)$ still assigns a different element of Δ^I to every element of Γ, and, given that Γ_O coincides with $\tau(\Lambda, \Gamma_V)$, this implies that different terms in $\tau(\Lambda, \Gamma_V)$ are interpreted as different objects in Δ_O^I, i.e., we enforce the unique name assumption on terms. Formally, this means that I is such that:

- for each $a \in \Gamma_V$: $a^I \in \Delta_V^I$,
- for each $a \in \Gamma_O$, i.e., for each $a \in \tau(\Lambda, \Gamma_V)$: $a^I \in \Delta_O^I$,
- for each $a, b \in \Gamma$, $a \neq b$ implies $a^I \neq b^I$.

The syntax and the semantics of a *DL-Lite*$_\mathcal{A}$ TBox, ABox and UCQ, introduced in the previous section, do not need to be modified. In particular, from the point of view of the semantics of queries, the notion of certain answers is exactly the same as the one presented in Section 2.4.

We can now turn our attention to the problem of specifying mapping assertions linking the data at the sources to the objects in the ontology. In the following, we make the following assumptions:

- As we said before, we assume that all value constants stored in $\mathcal{DB}$ belong to Γ_V, and that the data sources are wrapped into a relational database $\mathcal{DB}$ (constituted by the relational schema, and the extensions of the relations), so that we can query such data by using SQL.
- As mentioned in the introduction, the database $\mathcal{DB}$ is independent from the ontology; in other words, our aim is to link to the ontology a collection of data that exist autonomously, and have not been necessarily structured with the purpose of storing the ontology instances.
- $ans(\varphi, \mathcal{DB})$ denotes the set of tuples (of the arity of φ) of value constants returned as the result of the evaluation of the SQL query φ over the database $\mathcal{DB}$.

With these assumptions in place, to actually realize the link between the data and the ontology, we adapt principles and techniques from the literature on data integration [19]. In particular, we use the notion of *mappings* as described below.

A *DL-Lite*$_\mathcal{A}$ *ontology with mappings* is characterized by a triple $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ such that:

- $\mathcal{T}$ is a *DL-Lite*$_\mathcal{A}$ TBox;

- $\mathcal{DB}$ is a relational database;
- $\mathcal{M}$ is a set of *mapping assertions*, partitioned into two sets, $\mathcal{M}_t$ and $\mathcal{M}_a$, where:
  - $\mathcal{M}_t$ is a set of so-called *typing mapping assertions*, each one of the form

    $$\Phi \rightsquigarrow T_i$$

    where Φ is a query of arity 1 over $\mathcal{DB}$ denoting the projection of one relation over one of its columns, and T_i is one of the *DL-Lite*$_\mathcal{A}$ data types;
  - $\mathcal{M}_a$ is a set of *data-to-object mapping assertions* (or simply mapping assertions), each one of the form

    $$\Phi \rightsquigarrow \Psi$$

    where Φ is an arbitrary SQL query of arity $n > 0$ over $\mathcal{DB}$, Ψ is a conjunctive query over $\mathcal{T}$ of arity $n' > 0$ without non-distinguished variables, that possibly involves *variable terms*. A variable term is a term of the same form as the object terms introduced above, with the difference that variables appear as argument of the function. In other words, a variable term has the form $f(\boldsymbol{z})$, where f is a function symbol in Λ of arity m, and $\boldsymbol{z}$ denotes an m-tuple of variables.

We briefly comment on the assertions in $\mathcal{M}$ as defined above. Typing mapping assertions are used to assign appropriate types to constants in the relations of $\mathcal{DB}$. Basically, these assertions are used for interpreting the values stored in the database in terms of the types used in the ontology, and their usefulness is evident in all cases where the types in the data sources do not directly correspond to the types used in the ontology. Data-to-object mapping assertions, on the other hand, are used to map data in the database to instances of concepts, roles, and attributes in the ontology.

We next give an example of *DL-Lite*$_\mathcal{A}$ ontology with mappings.

Example 8. Let $\mathcal{DB}$ be the database constituted by a set of relations with the following signature:

$$D_1[\texttt{SSN}:\textbf{STRING},\texttt{PROJ}:\textbf{STRING},\ \texttt{D}:\textbf{DATE}],$$
$$D_2[\texttt{SSN}:\textbf{STRING},\texttt{NAME}:\textbf{STRING}],$$
$$D_3[\texttt{CODE}:\textbf{STRING},\texttt{NAME}:\textbf{STRING}],$$
$$D_4[\texttt{CODE}:\textbf{STRING},\texttt{SSN}:\textbf{STRING}]$$

We assume that, from the analysis of the above data sources, the following meaning of the above relations has been derived. Relation D_1 stores tuples (s, p, d), where s and p are strings and d is a date, such that s is the social security number of a temporary employee, p is the name of the project s/he works for (different projects have different names), and d is the ending date of the employment. Relation D_2 stores tuples (s, n) of strings consisting of the social security number s of an employee and her/his name n. Relation D_3 stores tuples (c, n) of strings

consisting of the code c of a manager and her/his name n. Finally, relation D_4 relates managers' code with their social security number.

A possible extension for the above relations is given by the following sets of tuples:

$$D_1 = \{(20903, Tones, 25\text{-}09\text{-}05)\}$$
$$D_2 = \{(20903, Rossi), (55577, White)\}$$
$$D_3 = \{(X11, White), (X12, Black)\}$$
$$D_4 = \{(X11, 29767)\}$$

Now, let $\Lambda = \{\mathbf{pers}, \mathbf{proj}, \mathbf{mgr}\}$ be a set of function symbols, where **pers**, **proj** and **mgr** are function symbols of arity 1. Consider the *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ such that $\mathcal{T}$ is the TBox of Example 1, and $\mathcal{M} = \mathcal{M}_t \cup \mathcal{M}_a$, where $\mathcal{M}_t$ is as follows:

$$\begin{array}{ll}
M_{t_1} : \texttt{SELECT SSN FROM } D_1 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_2} : \texttt{SELECT SSN FROM } D_2 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_3} : \texttt{SELECT CODE FROM } D_3 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_4} : \texttt{SELECT CODE FROM } D_4 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_5} : \texttt{SELECT PROJ FROM } D_1 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_6} : \texttt{SELECT NAME FROM } D_2 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_7} : \texttt{SELECT NAME FROM } D_3 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_8} : \texttt{SELECT SSN FROM } D_4 & \rightsquigarrow \texttt{ xsd:string} \\
M_{t_9} : \texttt{SELECT D FROM } D_1 & \rightsquigarrow \texttt{ xsd:date}
\end{array}$$

and $\mathcal{M}_a$ is as follows:

$$\begin{array}{lll}
M_{m_1} : \texttt{SELECT SSN,PROJ,D} & \rightsquigarrow & \mathit{tempEmp}(\mathbf{pers}(\texttt{SSN})), \\
\texttt{FROM } D_1 & & \mathit{WORKS\text{-}FOR}(\mathbf{pers}(\texttt{SSN}), \mathbf{proj}(\texttt{PROJ})), \\
& & \mathsf{ProjName}(\mathbf{proj}(\texttt{PROJ}), \texttt{PROJ}), \\
& & \mathsf{until}(\mathbf{pers}(\texttt{SSN}), \texttt{D}) \\
M_{m_2} : \texttt{SELECT SSN,NAME} & \rightsquigarrow & \mathit{employee}(\mathbf{pers}(\texttt{SSN})), \\
\texttt{FROM } D_2 & & \mathsf{PersName}(\mathbf{pers}(\texttt{SSN}), \texttt{NAME}) \\
M_{m_3} : \texttt{SELECT SSN, NAME} & \rightsquigarrow & \mathit{manager}(\mathbf{pers}(\texttt{SSN})), \\
\texttt{FROM } D_3, D_4 & & \mathsf{PersName}(\mathbf{pers}(\texttt{SSN}), \texttt{NAME}) \\
\texttt{WHERE } D_3\texttt{.CODE=}D_4\texttt{.CODE} & & \\
M_{m_4} : \texttt{SELECT CODE, NAME} & \rightsquigarrow & \mathit{manager}(\mathbf{mgr}(\texttt{CODE})), \\
\texttt{FROM } D_3 & & \mathsf{PersName}(\mathbf{mgr}(\texttt{CODE}), \texttt{NAME}) \\
\texttt{WHERE CODE NOT IN} & & \\
\quad \texttt{(SELECT CODE FROM } D_4\texttt{)} & &
\end{array}$$

We briefly comment on the data-to-ontology mapping assertions in $\mathcal{M}_a$. M_{m_1} maps every tuple (s, p, d) in D_1 to a temporary employee $\mathbf{pers}(s)$ with name p, working until d for project $\mathbf{proj}(p)$. M_{m_2} maps every tuple (s, n) in D_2 to an employee $\mathbf{pers}(s)$ with name n. M_{m_3} and M_{m_4} tell us how to map data in D_3 and D_4 to managers and their name in the ontology. Note that, if D_4 provides the social security number s of a manager whose code is in D_3, then we use the

social security number to form the corresponding object term, i.e., the object term has the form $\mathbf{pers}(s)$. If D_4 does not provide such information, then we use an object term of the form $\mathbf{mgr}(c)$ to denote the corresponding instance of the concept *manager*. □

In order to define the semantics of a *DL-Lite*$_\mathcal{A}$ ontology with mappings, we need to define when an interpretation *satisfies an assertion in* $\mathcal{M}$ *with respect to a database* $\mathcal{DB}$. To this end, we make use of the notion of ground instance of a formula. Let $\Psi(\boldsymbol{x})$ be a formula over a *DL-Lite*$_\mathcal{A}$ TBox with n distinguished variables $\boldsymbol{x}$, and let $\boldsymbol{v}$ a tuple of value constants of arity n. Then the ground instance $\Psi[\boldsymbol{x}/\boldsymbol{v}]$ of $\Psi(\boldsymbol{x})$ is the formula obtained by substituting every occurrence of x_i with v_i (for $i \in \{1, .., n\}$) in $\Psi(\boldsymbol{x})$. We are now ready to define when an interpretation satisfies a mapping assertion:

- Let m_t be an assertion in $\mathcal{M}_t$ of the form $\Phi \rightsquigarrow T_i$. We say that the interpretation I satisfies m_t with respect to a database $\mathcal{DB}$, if for every $v \in ans(\Phi, \mathcal{DB})$, we have that $v \in val(T_i)$.
- Let m_a be an assertion in $\mathcal{M}_a$ of the form

$$\Phi(\boldsymbol{x}) \rightsquigarrow \Psi(\boldsymbol{t}, \boldsymbol{y})$$

  where $\boldsymbol{x}$ and $\boldsymbol{y}$ are variables, $\boldsymbol{y} \subseteq \boldsymbol{x}$ and $\boldsymbol{t}$ are variable terms of the form $f(\boldsymbol{z})$, $f \in \Lambda$ and $\boldsymbol{z} \subseteq \boldsymbol{x}$.
  We say that I satisfies m_a with respect to a database $\mathcal{DB}$, if for every tuple of values $\boldsymbol{v}$ such that $\boldsymbol{v} \in ans(\Phi, \mathcal{DB})$, and for each ground atom X in $\Psi[\boldsymbol{x}/\boldsymbol{v}]$, we have that:
  - if X has the form $A(s)$, then $s^I \in A^I$;
  - if X has the form $D(s)$, then $s^I \in D^I$;
  - if X has the form $P(s_1, s_2)$, then $(s_1^I, s_2^I) \in P^I$;
  - if X has the form $U_C(s_1, s_2)$, then $(s_1^I, s_2^I) \in U_C^I$.

Finally, we say that an interpretation $I = (\Delta^I, \cdot^I)$ is a *model* of $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ if:

- I is a model of $\mathcal{T}$;
- I satisfies $\mathcal{M}$ with respect to $\mathcal{DB}$, i.e., satisfies every assertion in $\mathcal{M}$ with respect to $\mathcal{DB}$.

We denote as $Mod(\mathcal{O}_m)$ the set of models of $\mathcal{O}_m$, and we say that a *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m$ is satisfiable if $Mod(\mathcal{O}_m) \neq \emptyset$.

Example 9. One can easily verify that the ontology with mappings $\mathcal{O}_m$ of Example 8 is satisfiable. □

Note that, as we said in the introduction, the mapping mechanism described above nicely deals with the fact that the database $\mathcal{DB}$ and the ontology $\mathcal{O}_m$ are based on different semantical assumptions. Indeed, the semantics of $\mathcal{DB}$ follows the so-called "closed world assumption" [28], which intuitively sanctions that every fact that is not explicitly stored in the database is false. On the contrary,

the semantics of $\mathcal{O}_m$ is open, in the sense that nothing is assumed about the facts that do not appear explicitly in the ABox. In a mapping assertion of the form $\Phi \rightsquigarrow \Psi$, the closed semantics of $\mathcal{DB}$ is taken into account by the fact that Φ is evaluated as a standard relational query over the database $\mathcal{DB}$, while the open semantics of $\mathcal{O}_m$ is reflected by the fact that mappings assertions are interpreted as "material implication" in logic. It is well known that a material implication of the form $\Phi \rightsquigarrow \Psi$ imposes that every tuple of Φ contribute to the answers to Ψ, leaving open the possibility of additional tuples satisfying Ψ.

Let Q denote a UCQ expressed in terms of the TBox $\mathcal{T}$ of $\mathcal{O}_m$. We call *certain answers to Q posed over $\mathcal{O}_m$* the set of n-tuples of terms in Γ, denoted $Q^{\mathcal{O}_m}$, that is defined as follows:

$$Q^{\mathcal{O}_m} = \{\boldsymbol{t} \mid \boldsymbol{t}^I \in Q^I, \forall I \in Mod(\mathcal{O}_m)\}$$

Clearly, given an ontology with mappings and a query Q posed in terms of $\mathcal{T}$, query answering is the problem of computing the certain answers to Q.

4 Overview of the reasoning method

Our goal in the next sections is to illustrate a method for both checking satisfiability, and query answering in *DL-Lite*$_\mathcal{A}$ ontologies with mappings. In this section, we present an overview of our reasoning method, by concentrating in particular on the task of query answering.

The simplest way to tackle reasoning over a *DL-Lite*$_\mathcal{A}$ ontology with mappings is to use the mappings to produce an actual ABox, and then reasoning on the ontology constituted by the ABox and the original TBox, applying the techniques described in Section 2.4. We call such approach "bottom-up". However, such a bottom-up approach requires to actually build the ABox starting from the data at the sources, thus somehow duplicating the information already present in the data sources. To avoid such redundancy, we propose an alternative approach, called "top-down", which essentially keeps the ABox virtual.

We sketch out the main ideas of both approaches below. As we said before, we refer in particular to query answering, but similar considerations hold for satisfiability checking too. Before delving into the discussion, we define the notions of *split version of an ontology* and of *virtual ABox*, which will be useful in the sequel.

4.1 Splitting the mapping

Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a *DL-Lite*$_\mathcal{A}$ ontology with mappings as defined in the previous section. We show how to compute the *split version* of $\mathcal{O}_m$, that is characterized by a particularly "friendly form". Specifically, we denote as $\mathsf{Split}(\mathcal{O}_m) = \langle \mathcal{T}, \mathcal{M}', \mathcal{DB} \rangle$ a new ontology with mappings that is obtained from $\mathcal{O}_m$, by constructing $\mathcal{M}'$ as follows:

1. all typing assertions in $\mathcal{M}$ are also in $\mathcal{M}'$;

2. for each mapping assertion $\Phi \rightsquigarrow \Psi \in \mathcal{M}$, and for each atom $X \in \Psi$, the mapping assertion $\Phi' \rightsquigarrow X$ is in $\mathcal{M}'$, where Φ' is the projection of Φ over the variables occurring in X.

Example 10. Consider the ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ of Example 8. By splitting the mappings as described above, we obtain the ontology $\mathsf{Split}(\mathcal{O}_m) = \langle \mathcal{T}, \mathcal{M}', \mathcal{DB} \rangle$ such that $\mathcal{M}'$ contains all typing assertions in $\mathcal{M}$ and contains furthermore the following split mapping assertions:

$M_{m_{11}}$: `SELECT SSN FROM` D_1 $\rightsquigarrow$ *tempEmp*(**pers**(SSN))

$M_{m_{12}}$: `SELECT SSN, PROJ FROM` D_1 $\rightsquigarrow$ *WORKS-FOR*(**pers**(SSN), **proj**(PROJ))

$M_{m_{13}}$: `SELECT PROJ FROM` D_1 $\rightsquigarrow$ ProjName(**proj**(PROJ), PROJ)

$M_{m_{14}}$: `SELECT SSN,D FROM` D_1 $\rightsquigarrow$ until(**pers**(SSN), D)

$M_{m_{21}}$: `SELECT SSN FROM` D_2 $\rightsquigarrow$ *employee*(**pers**(SSN))

$M_{m_{22}}$: `SELECT SSN,NAME FROM` D_2 $\rightsquigarrow$ PersName(**pers**(SSN), NAME)

$M_{m_{31}}$: `SELECT SSN FROM` D_3, D_4 `WHERE` D_3`.CODE=`D_4`.CODE` $\rightsquigarrow$ *manager*(**pers**(SSN))

$M_{m_{32}}$: `SELECT SSN, NAME FROM` D_3, D_4 `WHERE` D_3`.CODE=`D_4`.CODE` $\rightsquigarrow$ PersName(**pers**(SSN), NAME)

$M_{m_{41}}$: `SELECT CODE FROM` D_3 `WHERE CODE NOT IN (SELECT CODE FROM` D_4`)` $\rightsquigarrow$ *manager*(**mgr**(CODE))

$M_{m_{42}}$: `SELECT CODE, NAME FROM` D_3 `WHERE CODE NOT IN (SELECT CODE FROM` D_4`)` $\rightsquigarrow$ PersName(**mgr**(CODE), NAME)

□

The relationship between an ontology with mappings and its split version is characterized by the following theorem.

Proposition 1. *Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a DL-Lite$_\mathcal{A}$ ontology with mappings. Then, we have that:*

$$Mod(\mathsf{Split}(\mathcal{O}_m)) = Mod(\mathcal{O}_m).$$

Proof. The result follows straightforwardly from the syntax and the semantics of the mappings. □

The theorem essentially tells us that every ontology with mappings is logically equivalent to the corresponding split version. Therefore, given any arbitrary *DL-Lite*$_\mathcal{A}$ ontology with mappings, we can always reduce it to its split version. Moreover, such a reduction has PTIME complexity in the size of the mappings and does not depend on the size of the data. This allows for assuming, from now on, to deal only with split versions of *DL-Lite*$_\mathcal{A}$ ontologies with mappings.

4.2 Virtual ABox

In this subsection we introduce the notion of virtual ABox. Intuitively, given a *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$, the virtual ABox corresponding to $\mathcal{O}_m$ is the ABox whose assertions are computed by "applying" the mapping assertions starting from the data in $\mathcal{DB}$. Note that in our method we do not explicitly build the virtual ABox. However, this notion will be used in the technical development presented in the sequel of the paper.

Definition 1. *Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a DL-Lite$_\mathcal{A}$ ontology with mappings, and let M be a mapping assertion in $\mathcal{M}$ of the form $M = \Phi \rightsquigarrow X$. We call* virtual ABox generated by M from $\mathcal{DB}$ *the following set of assertions:*

$$\mathcal{A}(M, \mathcal{DB}) = \{X[\boldsymbol{x}/\boldsymbol{v}] \mid \boldsymbol{v} \in ans(\Phi, \mathcal{DB})\},$$

where $\boldsymbol{v}$ and Φ are of arity n, and, as we said before, $X[\boldsymbol{x}/\boldsymbol{v}]$ denotes the ground atom obtained from $X(\boldsymbol{x})$ by substituting the n-tuple of variables $\boldsymbol{x}$ with the n-tuple of constants $\boldsymbol{v} \in \Gamma_V^n$. Moreover, the virtual ABox for $\mathcal{O}_m$, *denoted $\mathcal{A}(M, \mathcal{DB})$, is the set of assertions*

$$\mathcal{A}(\mathcal{M}, \mathcal{DB}) = \{\mathcal{A}(M, \mathcal{DB}) \mid M \in \mathcal{M}\}.$$

Notice that $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ is an ABox over the constants $\Gamma = \Gamma_V \cup \tau(\Lambda, \Gamma)$, as shown by the following example.

Example 11. Let $\mathsf{Split}(\mathcal{O}_m)$ be the *DL-Lite*$_\mathcal{A}$ ontology with split mappings of Example 10. Consider in particular the mappings $M_{m_{21}}$, $M_{m_{22}}$. Suppose we have $D_2 = \{(20903, Rossi), (55577, White)\}$ in the database $\mathcal{DB}$. Then, the sets of assertions $\mathcal{A}(M_{m_{21}}, \mathcal{DB})$, $\mathcal{A}(M_{m_{22}}, \mathcal{DB})$ are as follows:

$\mathcal{A}(M_{m_{21}}, \mathcal{DB}) = \{employee(\mathbf{pers}(20903)), employee(\mathbf{pers}(55577))\}$
$\mathcal{A}(M_{m_{22}}, \mathcal{DB}) = \{\mathsf{PersName}(\mathbf{pers}(20903), Rossi), \mathsf{PersName}(\mathbf{pers}(55577), White)\}$

□

By proceeding in the same way for each mapping assertion in $\mathcal{M}$, we can easily obtain the whole virtual ABox for $\mathcal{O}_m$.

Virtual ABoxes allow for expressing the semantics of *DL-Lite*$_\mathcal{A}$ ontologies with mappings in terms of the semantics of *DL-Lite*$_\mathcal{A}$ ontologies as follows:

Proposition 2. *If $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ is a DL-Lite$_\mathcal{A}$ ontology with mappings, then*

$$Mod(\mathcal{O}_m) = Mod(\langle \mathcal{T}, \mathcal{A}(\mathcal{M}, \mathcal{DB}) \rangle).$$

Proof. Trivial, from the definition. □

Now that we have introduced virtual ABoxes, we discuss in more detail both the bottom-up and the top-down approach.

4.3 A bottom-up approach

The proposition above suggests an obvious, and "naive", bottom-up algorithm to answer queries over a satisfiable *DL-Lite*$_\mathcal{A}$ ontology $O_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ with mappings, which we describe next. First, we materialize the virtual ABox for $\mathcal{O}_m$, i.e., we compute $\mathcal{A}(\mathcal{M}, \mathcal{DB})$. Second, we apply to the *DL-Lite*$_\mathcal{A}$ ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A}(\mathcal{M}, \mathcal{DB}) \rangle$, the algorithms for query answering, briefly described in Section 2.4.

Unfortunately, this approach has the following drawbacks. First, the time complexity of the proposed algorithm is PTime in the size of the database, since the generation of the virtual ABox is by itself a PTime process. Second, since the database is independent of the ontology, it may happen that, during the lifetime of the ontology with mappings, the data it contains are modified. This would clearly require to set up a mechanism for keeping the virtual ABox up-to-date with respect to the database evolution, similarly to what happens in data warehousing. Thus, next, we propose a different approach (called "top-down"), which uses an algorithm that avoids materializing the virtual ABox, but, rather, takes into account the mapping specification *on-the-fly*, during reasoning. In this way, we can both keep the computational complexity of the algorithm low, which turns out to be the same of the query answering algorithm for ontologies without mappings (i.e., in LogSpace), and avoid any further procedure for data refreshment.

4.4 A top-down approach

While the bottom-up approach described in the previous subsection is only of theoretical interest, we now present an overview of our top-down approach to query answering.

Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a *DL-Lite*$_\mathcal{A}$ ontology with split mappings, and let Q be a UCQ over $\mathcal{O}_m$. According to the top-down approach, query answering is constituted by three steps, called reformulation, unfolding, and evaluation, respectively.

- **Reformulation.** In this step, we compute the perfect reformulation $Q' = \mathsf{PerfectRef}(Q, \mathcal{T})$ of the original query Q, according to what we said in Section 2.4. Q' is a first-order logic query satisfying the following property: the certain answers to Q with respect to $\mathcal{O}_m$ coincide with the set of tuples computed by evaluating Q' over $db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))$[9], i.e., the database representing $\mathcal{A}(\mathcal{M}, \mathcal{DB})$.

[9] The function *db* is defined in Section 2.

- **Unfolding.** Instead of materializing $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ and evaluating Q' $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ (as in the bottom-up approach), we "unfold" Q' according to $\mathcal{M}$, i.e., we compute a new query Q'', which is an SQL over the source relations. As we will show in Section 6, this computation is done by using logic programming techniques, and allows us to get rid of $\mathcal{M}$, in the sense that the set of tuples computed by evaluating Q'' over the sources coincides with the set of tuples computed by evaluating Q' over $db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))$.
- **Evaluation.** The evaluation step consists simply in delegating the evaluation of Q'' over the database $\mathcal{DB}$ to the DBMS managing such database.

Example 12. Consider the ontology $\mathsf{Split}(\mathcal{O}_m)$ of Example 10, and assume it is satisfiable. The mapping assertions in $\mathcal{M}'$ of $\mathsf{Split}(\mathcal{O}_m)$ can be encoded in the following portion of a logic program (see Section 6):

$$
\begin{array}{ll}
\mathit{tempEmp}(\mathbf{pers}(s)) & \leftarrow \mathit{Aux}_{11}(s) \\
\mathit{WORKS\text{-}FOR}(\mathbf{pers}(s), \mathbf{proj}(p)) & \leftarrow \mathit{Aux}_{12}(s, p) \\
\mathsf{ProjName}(\mathbf{proj}(p), p) & \leftarrow \mathit{Aux}_{13}(p) \\
\mathsf{until}(\mathbf{pers}(s), d) & \leftarrow \mathit{Aux}_{14}(s, d) \\
\mathit{employee}(\mathbf{pers}(s)) & \leftarrow \mathit{Aux}_{21}(s) \\
\mathsf{PersName}(\mathbf{pers}(s), n) & \leftarrow \mathit{Aux}_{22}(s, n) \\
\mathit{manager}(\mathbf{pers}(s)) & \leftarrow \mathit{Aux}_{31}(s) \\
\mathsf{PersName}(\mathbf{pers}(s), n) & \leftarrow \mathit{Aux}_{32}(s, n) \\
\mathit{manager}(\mathbf{mgr}(c)) & \leftarrow \mathit{Aux}_{41}(c) \\
\mathsf{PersName}(\mathbf{mgr}(c), n) & \leftarrow \mathit{Aux}_{42}(c, n)
\end{array}
$$

where Aux_{ij} is a suitable predicate denoting the result of the evaluation over $\mathcal{DB}$ of the query $\Phi_{m_{ij}}$ in the left-hand side of the mapping $M_{m_{ij}}$ (note that for different Aux_{ih} and Aux_{ik}, we may have $\Phi_{m_{ih}}$ equal to $\Phi_{m_{ik}}$). Now, let

$$q(x) \leftarrow \mathit{WORKS\text{-}FOR}(x, y)$$

be the query discussed in Example 7. As we saw in Section 2, its reformulation $Q' = \mathsf{PerfectRef}(q, \mathcal{T})$ is:

$$
\begin{array}{l}
Q'(x) \leftarrow \mathit{WORKS\text{-}FOR}(x, y) \\
Q'(x) \leftarrow \mathsf{until}(x, y) \\
Q'(x) \leftarrow \mathit{tempEmp}(x) \\
Q'(x) \leftarrow \mathit{employee}(x) \\
Q'(x) \leftarrow \mathit{manager}(x)
\end{array}
$$

In order to compute the unfolding of Q', we unify each of its atoms in all possible ways with the left-hand side of the mapping assertions in $\mathcal{M}'$, and we obtain the following *partial evaluation* of Q':

$$
\begin{array}{l}
q(\mathbf{pers}(s)) \leftarrow \mathit{Aux}_{12}(s, p) \\
q(\mathbf{pers}(s)) \leftarrow \mathit{Aux}_{14}(s, d) \\
q(\mathbf{pers}(s)) \leftarrow \mathit{Aux}_{11}(s) \\
q(\mathbf{pers}(s)) \leftarrow \mathit{Aux}_{21}(s) \\
q(\mathbf{pers}(s)) \leftarrow \mathit{Aux}_{31}(s, n) \\
q(\mathbf{mgr}(c)) \leftarrow \mathit{Aux}_{41}(c, n)
\end{array}
$$

From the above formulation, it is now possible to derive the corresponding SQL query Q'' that can be directly issued over the database $\mathcal{DB}$:

```
SELECT CONCAT(CONCAT('pers (',SSN),')')
FROM D1
UNION
SELECT CONCAT(CONCAT('pers (',SSN),')')
FROM D2
UNION
SELECT CONCAT(CONCAT('pers (',SSN),')')
FROM D3, D4
WHERE D3.CODE=D4.CODE
UNION
SELECT CONCAT(CONCAT('mgr (',CODE),')')
FROM D3
WHERE CODE NOT IN (SELECT CODE FROM D4)
```

□

In the next two sections we delve into the details of our top-down method for reasoning about ontologies with mappings. In particular, in Section 5 we deal with the unfolding step, whereas in Section 6 we present the complete algorithms for both satisfiablity checking and query answering, and we discuss their formal properties. In both sections, we assume to deal only with ontologies with split mappings.

5 Dealing with mappings

As we saw in the previous section, the unfolding step is one of the ingredients of our top-down method for reasoning about ontologies with mappings. The goal of this section is to illustrate the technique we use to perform such a step.

Suppose we are given a *DL-Lite*$_{\mathcal{A}}$ ontology with split mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ and an UCQ Q over $\mathcal{O}_m$. The purpose of "unfolding" Q according to $\mathcal{M}$, is to compute a new query Q' satisfying the following properties:

1. Q' is a query (in particular, an SQL query) over the source relations,
2. the set of tuples computed by evaluating Q' over the data sources coincides with the set of tuples computed by evaluating Q over $db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))$.

From the above specification, it is clear that the unfolding step is crucial for avoiding materializing the virtual ABox.

The method we use for carrying out the unfolding step is based on logic programming notions [20]. The reason why we resort to logic programming is that mapping assertions are indeed similar to (simple forms of) rules of a logic program. The connection between data integration mappings and logic programming has already been noticed in several papers (see, for example, [25]). Our case, however, differs from those addressed in such papers, for two main reasons:

- while most of the above works use Datalog rules for modeling mappings, our mapping assertions contain functional terms, and therefore they go beyond Datalog;
- we do not want to use the rules for directly accessing data. Instead, we aim at using the rules for coming up with the right queries to ship to the data sources. In this sense, we use the rules only "partially".

The fact that we use the rules only partially is the reason why we will make use of the notion of "partial evaluation" of a logic program. This notion, together with more general notions of logic programming, is introduced in the next subsection.

5.1 Relevant notions from logic programming

We briefly recall some basic notions from logic programming [20], upon which we build our unfolding technique. In particular, we exploit some crucial results on the partial evaluation [18] of logic programs given in [21], which we briefly recall below.

Definition 2. *A* definite program clause *is an expression of the form*

$$A \leftarrow W$$

where A is an atom, and W is a conjunction of atoms $A_1, \ldots, A_n$. The left-hand side of a clause is called its head*, whereas its right-hand side is called its* body*. Either the body or the head of the clause may be empty. When the body is empty, the clause is called* fact *(and the $\leftarrow$ symbol is in general omitted). When the head is empty, the clause is called a* definite goal*. A* definite program *is a finite set of definite program clauses.*

Notice that $A \leftarrow W$ has a first-order logic reading, which is represented by the following sentence:

$$\forall x_1, \cdots, \forall x_s (A \vee \neg W).$$

where $x_1, \ldots, x_s$ are all the variables occurring in W and A. This reading explains why a logic program clause is also called a rule.

From now on, when we talk about programs, program clauses and goals, we implicitly mean definite programs, definite program clauses and definite goals, respectively.

A well-known property of logic programs is that every definite program $\mathcal{P}$ has a *minimal model*, which is the intersection $M_{\mathcal{P}}$ of all Herbrand models for $\mathcal{P}$ [20]. Intuitively, the minimal model of $\mathcal{P}$ is the set of all positive ground facts (i.e., atomic formulae without variables) that are true in all the models of $\mathcal{P}$. We say that an atomic formula (or atom) containing no variable is *true* in a logic program $\mathcal{P}$ if it is true in the minimal model of $\mathcal{P}$.

Logic program clauses are used to derive formulae from other formulae. The notion of derivation is formalized by the following definition.

Definition 3. *If G is a goal of the form $\leftarrow A_1, \cdots, A_m, \cdots, A_k$, and C is a program clause $A \leftarrow B_1, \cdots, B_q$, then G' is* derived *from G and C through the selected atom A_m using the most general unifier*[10] *(mgu) θ if the following conditions hold:*

- *θ is an mgu of A_m and A, and*
- *G' is the goal*

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$$

where $(A_1, \cdots, A_n)\theta = A_1\theta, \cdots, A_n\theta$, and $A\theta$ is the atom obtained from A applying the substitution θ.

Next we define the notion of resultant. We actually present a simplified definition of this notion, which is sufficient for our purpose.

Definition 4. *A* resultant *is an expression of the form*

$$\mathcal{Q}_1 \leftarrow \mathcal{Q}_2$$

where Q_1 is a conjunction of atoms, and Q_2 (called the body of the resultant) is either absent or a conjunction of atoms.

The possible derivations of a goal using a program are represented by a special tree, called SLD-tree, which is defined next.

Definition 5. *(SLD-Tree [21]) Let $\mathcal{P}$ be a program and let G' be a goal with body G. Then, an* SLD-Tree of $\mathcal{P} \cup \{G'\}$ *is a tree satisfying the following conditions:*

- *each node is a resultant,*
- *the root node is $G_0\theta_0 \leftarrow G_0$, where $G_0 = G$, and θ_0 is the empty substitution,*
- *let $G\theta_0 \cdots \theta_i \leftarrow G_i$*[11] *be a node N at depth $i \geq 0$ such that G_i has the form $A_1, \ldots, A_m, \ldots, A_k$, and suppose that A_m is the atom selected in G_i. Then, for each program clause C of the form $A \leftarrow B_1, \cdots, B_q$ in $\mathcal{P}$ such that A_m and A are unifiable with mgu θ_{i+1}, the node N has a child*

$$G\theta_0\theta_1 \cdots \theta_{i+1} \leftarrow G_{i+1},$$

where the goal $\leftarrow G_{i+1}$ is derived from the goal $\leftarrow G_i$ and C through A_m using θ_{i+1}, i.e., G_{i+1} has the form $(A_1, \ldots, B_1, \ldots, B_q, \ldots, A_k)\theta_{i+1}$,
- *a node which is a resultant with an empty body has no children.*

We say that a branch of an SLD-tree is failing *if it ends in a node such that the selected atom does not unify with the head of any program clause. Moreover, we say that an SLD-Tree is* complete *if all its non-failing branches end in the empty clause. An SLD-tree that is not complete is called* partial.

Finally, given a node $Q\theta_0, \ldots, \theta_i \leftarrow Q_i$ at depth i, we say that the derivation of Q_i has length *i with* computed answer *θ, where θ is the restriction of $\theta_0, \cdots, \theta_i$ to the variables in the goal G'.*

[10] A unifier of two expressions is a substitution of their variables that makes such expressions equal. A most general unifer is a unifier with a minimal number of substitutions.

[11] The expression $\theta_1\theta_2 \cdots \theta_n$ denotes the composition of the substitutions $\theta_1, \ldots, \theta_n$.

Finally, we recall the definition of *partial evaluation* (PE for short) from [21]. The definition actually refers to two kinds of PE: the *PE of an atom in a program*, and the *PE of a program with respect to an atom*. Intuitively, to obtain a PE of an atom A in $\mathcal{P}$, one considers an SLD-tree T for $\mathcal{P} \cup \{\leftarrow A\}$, and chooses a *cut* in T. The PE of $\mathcal{P}$ with respect to A is defined as the union of the resultants that occur in the cut and do not fail in T.

Definition 6. *Let $\mathcal{P}$ be a program, A an atom, and $\mathcal{T}$ an SLD-tree for $\mathcal{P} \cup \{\leftarrow A\}$. Then,*

- *any set of nodes such that each non-failing branch of T contains exactly one of them is a* PE of A in $\mathcal{P}$*;*
- *the logic program obtained from $\mathcal{P}$ by replacing the set of clauses in $\mathcal{P}$ whose head contains A with a PE of A in $\mathcal{P}$ is a* PE of $\mathcal{P}$ with respect to A*.*

Note that, by definition, a PE of A in $\mathcal{P}$ is a set of resultant, while the PE of $\mathcal{P}$ with respect to A is a logic program.

Also, a well-known property of PE is that a program $\mathcal{P}$ and any PE of $\mathcal{P}$ with respect to any atom are procedurally equivalent, i.e., the minimal model of $\mathcal{P}$ and the minimal model of any PE of $\mathcal{P}$ with respect to any atom coincide.

5.2 The unfolding step

We are now ready to look into unfolding step for reasoning in *DL-Lite*$_\mathcal{A}$ ontologies with mappings. In particular, the goal is to define a function UnfoldDB, that, intuitively, takes as input a *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$, and a UCQ (possibly with inequalities) Q over $\mathcal{O}_m$, and returns a set of resultants describing

1. the queries to issue to $\mathcal{DB}$, and
2. the substitution to apply to the result in order to obtain the answer to Q.

In order to use logic programming based techniques for unfolding, we express the UCQ Q and the relevant information about $\mathcal{M}$ and $\mathcal{DB}$ in terms of a logic program, called the *program for Q and $\mathcal{O}_m$*.

In all this subsection, unless otherwise stated, we consider $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ to be a *DL-Lite*$_\mathcal{A}$ ontology with mappings, and Q to be a union of conjunctive queries over $\mathcal{O}_m$, possibly including inequalities.

Definition 7. *The* program for Q and $\mathcal{O}_m$*, denoted $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, is the logic program formed as follows:*[12]

1. *for each conjunctive query $(q(\boldsymbol{x}) \leftarrow Q') \in Q$, $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ contains the clause*

$$q(\boldsymbol{x}) \leftarrow \sigma(Q')$$

where $\sigma(\alpha)$ denotes the query obtained by replacing each $x \neq y$ in the body of α with the atom $Distinct(x, y)$, where $Distinct$ is an auxiliary binary predicate;

[12] We assume that the alphabet of $\mathcal{T}$ does not contain the predicate *Distinct*, and, for any i, does not contain the predicate Aux_i.

2. *for each mapping assertion* $m_k \in \mathcal{M}$ *of the form* $\Phi_k(\boldsymbol{x}) \rightsquigarrow p_k(\boldsymbol{t})$, $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ *contains the clause*

$$p_k(\boldsymbol{t}) \leftarrow Aux_k(\boldsymbol{x})$$

where Aux_k *is an auxiliary predicate associated to* m_k*, whose arity is the same as* Φ_k*;*
3. *for each* Φ_k *appearing in the left-hand side of a mapping assertion in* $\mathcal{M}$*, for each* $\boldsymbol{t} \in ans(\Phi_k, \mathcal{DB})$, $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ *contains the fact* $Aux_k(\boldsymbol{t})$*;*
4. *let* $\Gamma_{\mathcal{DB}}$ *be the set of all values appearing in* $\mathcal{DB}$*; then for each pair* t_1, t_2 *of distinct terms in* $\tau(\Lambda, \Gamma_{\mathcal{DB}}) \cup \Gamma_{\mathcal{DB}}$, $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ *contains the fact* $Distinct(t_1, t_2)$.

Intuitively, item (1) in the definition is used to represent the query Q in the logic program $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, with the proviso that all inequalities are expressed in terms of the predicate *Distinct*. Item (2) introduces one auxiliary predicates Aux_i for each Φ_i appearing in the left-hand side of the mapping assertion in $m_i \in \mathcal{M}$, and item (3) states that the extension of $Aux_i(\boldsymbol{x})$ coincides with $ans(\Phi_i, \mathcal{DB})$. Finally, item (4) is used to enforce the unique name assumptionin $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$.

The following two lemmas state formally the relationship between $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, $\mathcal{O}_m$ and Q. They essentially show that $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ is a faithful representation in logic programming of both $\mathcal{O}_m$ and Q.

Lemma 1. $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ *coincides with the projection over the alphabet of* $\mathcal{T}$ *of the minimal model of* $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$.

Proof. We first show that, for each tuple $\boldsymbol{t}$ of terms, if $X(\boldsymbol{t}) \in \mathcal{A}(\mathcal{M}, \mathcal{DB})$, then $X(\boldsymbol{t})$ is true in the minimal model $M_{\mathcal{P}}$ of $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$. Consider a tuple $\boldsymbol{t}$ such that $X(\boldsymbol{t}) \in \mathcal{A}(\mathcal{M}, \mathcal{DB})$. Thus, by construction of $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ we have that there exists a mapping $\Phi_k(\boldsymbol{x}) \rightsquigarrow X(\alpha)$ in $\mathcal{M}$, a tuple $\boldsymbol{t}'$ of values in $\Gamma_{\mathcal{DB}}$, and a substitution θ such that $\boldsymbol{t}' \in ans(\Phi_k, \mathcal{DB})$ and $\boldsymbol{t} = \alpha\theta$. But then, since $\boldsymbol{t}' \in ans(\Phi_k, \mathcal{DB})$, we have that $Aux_k(\boldsymbol{t}') \in \mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$. Moreover, since $\Phi_k(\boldsymbol{x}) \rightsquigarrow X(\alpha)$ is a mapping in $\mathcal{M}$, we have that $X(\alpha) \leftarrow Aux_k(\boldsymbol{x}) \in \mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$. Thus, θ is an *mgu* of $Aux_k(\boldsymbol{x})$ and $Aux_k(\boldsymbol{t}')$. Therefore, it is possible to derive $X(\boldsymbol{t})$ from $Aux_k(\boldsymbol{t}')$ and $X(\alpha) \leftarrow Aux_k(\boldsymbol{x})$ by using θ, and, by a well-known property of logic programming, $X(\boldsymbol{t})$ is true in $M_{\mathcal{P}}$.

Conversely, let $X(\boldsymbol{t})$ be true in the minimal model $M_{\mathcal{P}}$ of $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$. By following a similar line of reasoning as above, it can be easily shown that $X(\boldsymbol{t}) \in \mathcal{A}(\mathcal{M}, \mathcal{DB})$. □

Lemma 2. *For each tuple* $\boldsymbol{t} \in \tau(\Gamma_V, \Lambda) \cup \Gamma_V$*, we have that*

$\boldsymbol{t} \in ans(Q, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})))$ *if and only if* $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}) \cup \{\leftarrow q(\boldsymbol{t})\}$ *is unsatisfiable.*

Proof. The result follows directly from the previous lemma and the construction of $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$. □

We now illustrate how to compute a specific PE of the program $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ (where Q has the form $q(\boldsymbol{x}) \leftarrow \beta$) with respect to $q(\boldsymbol{x})$, denoted $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$. Such a PE is crucial for our development.

We first define the function $\mathsf{SLD\text{-}Derive}(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}))$ that takes as input $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, and returns a set S of resultants constituting a PE of $q(\boldsymbol{x})$ in $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, by constructs an SLD-Tree T for $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}) \cup \{\leftarrow q(\boldsymbol{x})\}$ as follows:

- it starts by selecting the atom $q(\boldsymbol{x})$,
- it continues by selecting the atoms whose predicates belong to the alphabet of $\mathcal{T}$, as long as possible;
- it stops the construction of a branch when no atom with predicate in the alphabet of $\mathcal{T}$ can be selected.

Note that the above definition implies that $\mathsf{SLD\text{-}Derive}(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}))$ returns the set S of resultants obtained by cutting T only at nodes whose body contains only atoms with predicate Aux_i or $Distinct$.

Second, we use $\mathsf{SLD\text{-}Derive}(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}))$ to define $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, a specific PE of $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ with respect to $q(\boldsymbol{x})$. $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ is obtained simply by dropping the clauses for q in $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, and replacing them by $S = \mathsf{SLD\text{-}Derive}(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}))$.

Obviously, since $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ is a PE of $\mathcal{P}$, the two programs are procedurally equivalent, i.e., for every atom A, A is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ if and only if A is true in $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$.

Let Q a UCQ of the form $q(\boldsymbol{x}) \leftarrow \beta$. We now define the function $spread_{\mathcal{O}_m}$ that takes as imput a resultant $q(\boldsymbol{x})\theta \leftarrow Q'$ in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, and returns an extended form of resultant $q(\boldsymbol{x})\theta \leftarrow Q''$ such that Q'' is a first-order query over $\mathcal{DB}$, which is obtained from Q' by proceeding as follows. At the beginning, Q'' has an empty body. Then, for each atom A in Q',

- if $A = Aux_k(\boldsymbol{x})$, it adds to Q' the query $\Phi_k(\boldsymbol{x})$; note that, by hypothesis, $\Phi_k(\boldsymbol{x})$ is an arbitrary first-order query with distinguished variables $\boldsymbol{x}$, that can be evaluated over $\mathcal{DB}$;
- if $A = Distinct(x_1, x_2)$, where x_1, x_2 have resp. the form $f_1(\boldsymbol{y}_1)$ and $f_2(\boldsymbol{y}_2)$, then:
  - if $f_1 \neq f_2$, then it does nothing,
  - otherwise, it adds to Q' the following conjunct:

$$\bigvee_{i \in \{1,\ldots,w\}} y_{1_i} \neq y_{2_i},$$

    where w is the arity of f_1.

  Note that in this case we obtain a disjunction of variables, which, again, can be obviously evaluated over a set of data sources $\mathcal{DB}$.

The next lemma establishes the relationship between the answers to the program $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, and the tuples that are answers in the queries over the data sources that are present in the mapping assertions. It essentially sayss that

every answer to the program $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ is "generated" by tuples in the data sources.

This lemma and the next theorems make use of a new notion, that we now introduce. We say that an atom $q(\boldsymbol{t})$ is *obtained from* $q(\boldsymbol{x})$ *and* θ *through* $\boldsymbol{t}'$ if $q(\boldsymbol{x})\theta = q(\boldsymbol{t})$, and all constants used in θ appear in $\boldsymbol{t}'$. For example, $q(f(2,3),4)$ is obtained from $q(x_1, x_2)$ and the substitution $\{x_1/f(2,3), x_2, 4\}$ through the tuple $(2,3,4)$.

Lemma 3. *Let Q a UCQ of the form $q(\boldsymbol{x}) \leftarrow \beta$. For each tuple $\boldsymbol{t} \in \tau(\Gamma_V, \Lambda) \cup \Gamma_V$, $q(\boldsymbol{t})$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ if and only if there is a resultant $q(\boldsymbol{x})\theta \leftarrow Q'$ in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ and a tuple $\boldsymbol{t}'$ in $\Gamma_{\mathcal{V}}$ such that $q(\boldsymbol{x})\theta = q(\boldsymbol{t})$, $q(\boldsymbol{t})$ is obtained from $q(\boldsymbol{x})$ and θ through $\boldsymbol{t}'$, $spread_{\mathcal{O}_m}(q(\boldsymbol{x})\theta \leftarrow Q') = (q(\boldsymbol{x})\theta \leftarrow Q'')$, and $\boldsymbol{t}' \in ans(Q'', \mathcal{DB})$.*

Proof. The if-direction is easy to prove. For the only-if-direction, if $q(\boldsymbol{t})$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, $q(\boldsymbol{t})$ can be derived using a resultant in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$. Let $q(\boldsymbol{x})\theta \leftarrow Q'$ be such a resultant in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, and let Q' have the form $A_1(\boldsymbol{x}_1), \cdots, A_n(\boldsymbol{x}_n)$. By construction, $A_i(\boldsymbol{x}_i)$ is either

- $Aux_{k_i}(\boldsymbol{x}_i)$, or
- $Distinct(\boldsymbol{x}_i)$, where $\boldsymbol{x}_i = (x_{i_1}, x_{i_2})$.

Suppose that A_i has predicate Aux_{k_i} for each $i \leq j$ whereas it has predicate $Distinct$ for $j < i \leq n$. By construction, $spread_{\mathcal{O}_m}(q(\boldsymbol{x})\theta \leftarrow Q') = (q(\boldsymbol{x})\theta \leftarrow Q'')$, with Q'' of the form:

$$\{\boldsymbol{x}, \cdots y_{i_{1_1}}, y_{i_{2_1}}, \cdots y_{i_{1_{w_i}}}, y_{i_{2_{w_i}}} \cdots \\ \mid \Phi_{k_1}(\boldsymbol{x}_1), \cdots, \Phi_{k_j}(\boldsymbol{x}_j), (\bigvee_{i \leq n} \quad (\bigvee_{h \in \{1,\ldots,w_i\}} y_{i_{1_h}} \neq y_{i_{2_h}}))\}$$

where $(\bigvee_{h \in \{1,\ldots,w_i\}} y_{i_{1_h}} \neq y_{i_{2_h}})$ occurs together with the corresponding distinguished variables $y_{i_{1_h}}, y_{i_{2_h}}$ if there is an atom $Distinct(x_{i_1}, x_{i_2})$ in q such that $x_{i_1} = f(\boldsymbol{y}_{i_1}), x_{i_2} = f(\boldsymbol{y}_{i_2})$ where f has arity w_i.

Now, let $\boldsymbol{t}$ be a tuple in $\tau(\Gamma_V, \Lambda) \cup \Gamma_V$. We show next that if $q(\boldsymbol{t})$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, then there is a tuple $\boldsymbol{t}'$ in $\Gamma_{\mathcal{V}}$ such that $q(\boldsymbol{x})\theta = Q(\boldsymbol{t})$, $q(\boldsymbol{t})$ is obtained from $q(\boldsymbol{x})$ and θ through $\boldsymbol{t}'$, and $\boldsymbol{t}' \in ans(Q'', \mathcal{DB})$. Suppose that $q(\boldsymbol{t})$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$. Then there exists θ^q such that $q(\boldsymbol{t}) = (A_1(\boldsymbol{x}_1), \cdots, A_n(\boldsymbol{x}_n))\theta^q$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$. This implies that there exist n facts F_i in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ such that $F_i = A_i\theta^q$ is true in $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$ for each $i = 1, \cdots, n$. But then, by construction:

- if $i \leq j$, then F_i has the form $Aux_{k_i}(\boldsymbol{t}_i)$, which by construction means that $\boldsymbol{t}_i \in ans(\Phi_{k_i}, \mathcal{DB})$;
- otherwise, F_i has the form $Distinct(\boldsymbol{t}_i)$, where $\boldsymbol{t}_i = (t_{i1}, t_{i2})$ and t_{i1}, t_{i2} are such that $t_{i1} \neq t_{i2}$.

By the above observations, one can easily verify that $\boldsymbol{t}' \in ans(Q'', \mathcal{DB})$. Indeed, for $i \leq j$ we have trivially that $\Phi_{k_i}(\boldsymbol{t}'_i)$ is true, whereas for $j < i \leq k$, we have that if $f_1 = f_2$, then $\boldsymbol{v_{i_1}} \neq \boldsymbol{v_{i_2}}$. Thus, since $q(\boldsymbol{x})\theta^q$ belongs to $\mathcal{PE}(Q, \mathcal{M}, \mathcal{DB})$, then $q(\boldsymbol{t})$ is obtained from $q(\boldsymbol{x})$ and θ^p through $\boldsymbol{t}'$, and we have proved the claim. □

Before presenting the function UnfoldDB, we need to make a further observation. The program $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$, being a faithful representation of Q and $\mathcal{O}_m$, contains also the facts regarding the predicates Aux_i and $Distinct$. Since we use partial evaluation techniques for computing the queries to issue to the data sources, we are interested in the program obtained from $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ by ignoring such facts. Formally, we define $\mathcal{P}(Q, \mathcal{M})$ to be the program obtained from $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ by eliminating all facts $Aux_k(\boldsymbol{t})$ and $Distinct(\boldsymbol{t})$. Notice that while $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ depends on the $\mathcal{DB}$, $\mathcal{P}(Q, \mathcal{M})$ does not. The next theorem shows that the programs $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ and $\mathcal{P}(Q, \mathcal{M})$ are equivalent with respect to partial evaluation.

Lemma 4. *SLD-Derive*$(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})) =$ *SLD-Derive*$(\mathcal{P}(Q, \mathcal{M}))$.

Proof. The proof follows from the observation that SLD-Derive$(\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}))$ constructs an SLD-Tree for $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB}) \cup \{\leftarrow q(\boldsymbol{x})\}$ by selecting only the atoms in the alphabet of $\mathcal{T}$, and that $\mathcal{P}(Q, \mathcal{M}, \mathcal{DB})$ and $\mathcal{P}(Q, \mathcal{M})$ coincide in the clauses containing atoms in the alphabet of $\mathcal{T}$. □

Now we are finally able to come back to the definition of UnfoldDB. As usual in this subsection, $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ is an ontology with mappings, and Q a union of conjunctive queries (possibly with inequalities) over $\mathcal{O}_m$. We define UnfoldDB$(Q, \mathcal{O}_m)$ as the function that takes as input Q and $\mathcal{O}_m$, and returns a set S' of resultants by proceeding as shown in Fig. 1.

Algorithm UnfoldDB$(Q, \mathcal{O}_m)$
Input: *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$
UCQ (possibly with inequalities) Q over $\mathcal{O}_m$
Output: set of resultants S'
build the program $\mathcal{P}(Q, \mathcal{M})$;
compute the set of resultants $S =$ SLD-Derive$(\mathcal{P}(Q, \mathcal{M}))$;
for each $ans\theta \leftarrow q \in S$ **do**
$S' \leftarrow spread_{\mathcal{O}_m}(ans\theta \leftarrow q)$;
return S'

Fig. 1. The Algorithm UnfoldDB

The next theorem shows the correctness of UnfoldDB, i.e., termination, soundeness and completeness.

Theorem 3. *For every UCQ Q of the form $q(\boldsymbol{x}) \leftarrow \beta$ and for every $\mathcal{O}_m$,* UnfoldDB$(Q, \mathcal{O}_m)$ *terminates, and for each tuple of constants $\boldsymbol{t}$ in $\tau(\Gamma_V, \Lambda) \cup \Gamma_V$ we have that:*

$\boldsymbol{t} \in ans(Q, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})))$ *if and only if*
$\exists (q(\boldsymbol{x})\theta \leftarrow Q'') \in$ UnfoldDB$(Q, \mathcal{O}_m)$ *such that* $q(\boldsymbol{x})\theta = q(\boldsymbol{t})$,
$q(\boldsymbol{t})$ *is obtained from* $q(\boldsymbol{x})$ *and* θ *through* $\boldsymbol{t}'$, *and* $\boldsymbol{t}' \in ans(Q'', \mathcal{DB})$.

Proof. Termination of UnfoldDB is immediate. Soundness and completeness can be directly proved by using the lemmas presented in this section. □

Note that the algorithm UnfoldDB described in Fig. 1 returns a set of resultants, called S'. This form of the algorithm was instrumental for proving the correctness of our method. However, from a practical point of view, the best choice is to translate these set of resultants into a suitable SQL query that can be issued on the data sources. Indeed, this is exactly what our current implementation does. In particular, in the implementation, the final **for each** loop in the algorithm is replaced by a step that, starting from S, builds an SLQ query that, once evaluated over the data sources, computes directly the answers of the original query Q. For the sake of space, we do not describe such a step here. We only note that the kind of SQL queries obtained with this method can be seen by looking at example 12 in Section 4.

We end this section by observing that UnfoldDB allows for completely forgetting about the mappings during query evaluation, by compiling them directly in the queries to be posed over the underlying database. Next we show that this crucial property allows for devising reasoning procedures that exploit, on one hand, the results on reasoning over *DL-Lite*$_\mathcal{A}$ ontologies, and, on the other hand, the ability of the underlying database of answering arbitrary first-order queries.

6 Reasoning over *DL-Lite*$_\mathcal{A}$ ontologies with mappings

Now that we have described the unfolding step, we are ready to illustrate the complete algorithms for both satisfiablity checking and query answering, and to discuss their formal properties. We deal with satisfiability checking first, and then we address query answering.

Both algorithms make use of several functions that were introduced in the previous sections, and that we recall here. In what follows, we refer to a *DL-Lite*$_\mathcal{A}$ ontology $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ with split mappings.

- The boolean function Violates takes as input the TBox $\mathcal{T}$, and computes a first-order query over $\mathcal{O}_m$ that intuitively looks for violations of functionality and disjointness assertions specified in the TBox $\mathcal{T}$.
- The function PerfectRef takes as input a UCQ Q over $\mathcal{O}_m$ and the TBox $\mathcal{T}$, and reformulates Q into a new query Q', which is still a UCQ and has the following property: answering Q' with respect to $\langle \emptyset, \mathcal{M}, \mathcal{DB} \rangle$ is the same as answering Q with respect to $\langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$.
- The function UnfoldDB is the one discussed in Section 5.

6.1 Satisfiability checking

In Fig. 2 we present the Algorithm Sat that checks the satisfiability of a *DL-Lite*$_\mathcal{A}$ ontology with mappings. More precisely, Sat($\mathcal{O}_m$) issues the call Violates($\mathcal{T}$) to compute the query Q^s, asking, for each functionality assertion in $\mathcal{T}$ and

each negative inclusion assertion in $cln(\mathcal{T})$, whether $db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))$ violates the assertion. Then, by calling $\mathsf{UnfoldDB}(Q^s, \mathcal{O}_m)$, that allows for "compiling" the knowledge represented by the mapping assertions, $\mathsf{Sat}(\mathcal{O}_m)$ computes the set of resultants S' as discussed in the previous section. After extracting from S' the union of queries Q', $\mathsf{Sat}(\mathcal{O}_m)$ evaluates Q' over $\mathcal{DB}$, and returns *true*, if and only if $ans(Q', \mathcal{DB}) = false$.

Algorithm Sat
Input: *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$
Output: *true* or *false*
$Q^s \leftarrow \mathsf{Violates}(\mathcal{T})$;
$S' \leftarrow \mathsf{UnfoldDB}(Q^s, \mathcal{O}_m)$;
$Q' \leftarrow false$;
for each $ans\theta \leftarrow q' \in S'$ **do**
$Q' \leftarrow Q' \cup \{q'\}$;
return not$(ans(Q', \mathcal{DB}))$

Fig. 2. The Algorithm Sat

We next show the correctness of Algorithm Sat.

Theorem 4. *Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a DL-Lite$_\mathcal{A}$ ontology with mappings. Then, $\mathsf{Sat}(\mathcal{O}_m)$ terminates, $\mathcal{O}_m$ is satisfiable if and only if $\mathsf{Sat}(\mathcal{O}_m) = true$.*

Proof. The termination of the algorithm follows from the termination of UnfoldDB.

Concerning the soundness and the completeness of the algorithm, by Proposition 2, we have that $\mathcal{O}_m$ is satisfiable if and only if $\mathcal{O} = \langle \mathcal{T}, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})) \rangle$ is unsatisfiable. Moreover, as discussed in Section 2.4, we have that $\mathcal{O} = \langle \mathcal{T}, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})) \rangle$ is unsatisfiable if and only if $ans(Q^s, db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))) = true$ where $Q^s = \mathsf{Violates}(\mathcal{T})$. Thus, in order to prove the theorem, it suffices to prove that:

$$(*) ans(Q^s, db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))) = true \text{ if and only if } ans(Q', \mathcal{DB}) = true,$$

where Q' is such that $Q' = \bigcup_{ans\theta \leftarrow q' \in S'} q'$ and $S' = \mathsf{UnfoldDB}(Q^s, \mathcal{O}_m)$.

Clearly, this concludes the proof, since $(*)$ follows straightforwardly from the correctness of UnfoldDB. □

6.2 Query answering

In Fig. 3 we present the algorithm Answer to answer UCQs posed over a *DL-Lite*$_\mathcal{A}$ ontology with mappings. Informally, the algorithm takes as input a *DL-Lite*$_\mathcal{A}$ ontology $\mathcal{O}_m$ with mappings and a UCQ Q over $\mathcal{O}_m$. If the ontology is not satisfiable, then it returns the set of all possible tuples of elements in $\Gamma_0 \cup \Gamma_\mathcal{V}$

denoted $AllTup(Q, \mathcal{O}_m)$, whose arity is the one of the query Q. Otherwise, it computes the perfect reformulation Q^p of Q, and then unfold Q^p by calling $\mathsf{UnfoldDB}(Q^p, \mathcal{O}_m)$ to compute the set of resultants S'. Then, for each resultant $\mathcal{Q}'$ in S', it extracts the conjunctive query in its body, evaluates it over $\mathcal{DB}$ and further processes the answers according to the substitution occurring in the head of $\mathcal{Q}'$.

Algorithm Answer
Input: *DL-Lite*$_\mathcal{A}$ ontology with mappings $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$,
UCQ Q over $\mathcal{O}_m$
Output: set of tuples R^s
if $\mathcal{O}_m$ is not satisfiable
then return $AllTup(Q, \mathcal{O}_m)$
else
$Q^p \leftarrow \bigcup_{q_i \in Q} \mathsf{PerfectRef}(q_i, \mathcal{T})$;
$S' \leftarrow \mathsf{UnfoldDB}(Q^p, \mathcal{O}_m)$;
$R^s \leftarrow \emptyset$;
for each $ans\theta \leftarrow q' \in S'$ **do**
$R^s \leftarrow R^s \cup ans(q', \mathcal{DB})\theta$;
return R^s

Fig. 3. Algorithm $\mathsf{Answer}(Q, \mathcal{O}_m)$

We next show the correctness of Algorithm Answer.

Theorem 5. *Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a DL-Lite$_\mathcal{A}$ ontology with mappings, and Q a union of conjunctive queries over $\mathcal{O}_m$. Then, $\mathsf{Answer}(Q, \mathcal{O}_m)$ terminates. Moreover, let R^s be the set of tuples returned by $\mathsf{Answer}(Q, \mathcal{O}_m)$, and let $\boldsymbol{t}$ be a tuple of elements in $\Gamma_0 \cup \Gamma_\mathcal{V}$. Then, $\boldsymbol{t} \in ans(Q, \mathcal{O}_m)$ if and only if $\boldsymbol{t} \in R^s$.*

Proof. The termination of the algorithm follows from the termination of the Algorithm PerfectRef and the function UnfoldDB.

Concerning the soundness and completeness of the Algorithm Answer, by Proposition 2, we have that: $Mod(\mathcal{O}_m) = Mod(\mathcal{O})$, where $\mathcal{O} = \langle \mathcal{T}, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})) \rangle$. Moreover, given a union of conjunctive queries Q, as discussed in Section 2.4, we have that $ans(Q, \mathcal{O}) = ans(Q^p, db(\mathcal{A}(\mathcal{M}, \mathcal{DB})))$, where $(Q^p) = \mathsf{PerfectRef}(Q)$. Then, since by definition, we have that:

- $ans(Q, \mathcal{O}) = \{\boldsymbol{t} \mid \boldsymbol{t}^I \in Q^I, I \in Mod(\mathcal{O})\}$, and
- $Q^{\mathcal{O}_m} = \{\boldsymbol{t} \mid \boldsymbol{t}^I \in Q^I, I \in Mod(\mathcal{O}_m)\}$,

it is easy to see that:

$$ans(Q, \mathcal{O}_m) = ans(Q^p, db(\mathcal{A}(\mathcal{M}, \mathcal{DB}))).$$

On the other hand, by construction, we have that:

$$R^s = \{\boldsymbol{t}'\theta \mid \boldsymbol{t}' \in ans(q', \mathcal{DB}), ans\theta \leftarrow q' \in S'\}$$

where S' is such that $S' = \mathsf{UnfoldDB}(Q^p, \mathcal{O}_m)$. Then, clearly, by the correctness of UnfoldDB, we obtain the claim. □

Note that the algorithm Answer reconstructs the result starting from the results obtained by evaluating the SQL queries q' over the database $\mathcal{DB}$. However, from a practical point of view, we can simply delegate such a reconstruction step to the SQL engine. Indeed, this is exactly what our current implementation does. In particular, in the implementation, the final **for each** loop in the algorithm is replaced by a step that, starting from S', builds an SQL query that, once evaluated over the data sources, computes directly the answers of the original query Q.

6.3 Computational complexity

We first study the complexity of UnfoldDB. Note that, in this section, we assume that the mappings in $\mathcal{M}$ involve SQL queries over the underlying database $\mathcal{DB}$, and such SQL queries belong to the class of first-order logic queries. So, queries in the left-hand side of our mapping assertions, are LogSpace with respect to the size of the data in $\mathcal{DB}$.[13]

Lemma 5. *Let $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$ be a DL-Lite$_\mathcal{A}$ ontology with mappings, and Q a UCQ over $\mathcal{O}_m$. The function* $\mathsf{UnfoldDB}(Q, \mathcal{O}_m)$ *runs in exponential time with respect to the size of Q, and in polynomial time with respect to the size of $\mathcal{M}$.*

Proof. Let Q be a UCQ, and let n be the total number of atoms in the body of all q's in Q. Moreover, let m be the number of mappings and let m_n be the maximum size of the body of mappings. The result follows immediately by considering the cost of each of the three steps of $\mathsf{UnfoldDB}(Q, \mathcal{O}_m)$:

1. The construction of $\mathcal{P}(Q, \mathcal{M})$ is clearly polynomial in n and m.
2. The computation of $\mathsf{SLD\text{-}Derive}(\mathcal{P}(Q, \mathcal{M}))$ builds first a tree of depth at most n such that each of its nodes has at most m children, and, second, it processes all the leaves of the tree to obtain the set S of resultants. By construction, this set has size $\mathcal{O}_m(m^n)$. Clearly, the overall computation has complexity $\mathcal{O}_m(m^n)$.
3. Finally, the application of the function $spread_{\mathcal{O}_m}$ to each element in S has complexity $\mathcal{O}_m(m^n \cdot m_n)$. □

[13] The assumption of dealing with SQL queries that are first-order logic queries allows for using the most common SQL constructs (except for few of them, e.g., the "groupby" construct). Obviously, our approach works also for arbitrary SQL queries. In such a case, the complexity of the overall approach is the complexity of evaluating such queries over the underlying database.

Based on the above property, we are able to establish the complexity of checking the satisfiability of a *DL-Lite*$_\mathcal{A}$ ontology with mappings and the complexity of answering UCQ over it.

Theorem 6. *Given a DL-Lite*$_\mathcal{A}$ *ontology with mappings* $\mathcal{O}_m = \langle \mathcal{T}, \mathcal{M}, \mathcal{DB} \rangle$, $\mathsf{Sat}(\mathcal{O}_m)$ *runs* LogSpace *in the size of* $\mathcal{DB}$ *(data complexity). Moreover, it runs in polynomial time in the size of* $\mathcal{M}$*, and in polynomial time in the size of* $\mathcal{T}$*.*

Proof. The claim is a consequence of the results discussed in Section 2.4, i.e., the fact that $\mathcal{O}_m$ is satisfiable if and only if $ans(\mathsf{Violates}(\mathcal{T}), db(\mathcal{A}(\mathcal{M}, \mathcal{DB})))$:

1. $\mathsf{Violates}(\mathcal{T})$ returns a union of queries Q^s over $db(\mathcal{A}(\mathcal{M}, \mathcal{D}))$ whose size is polynomial in the size of $\mathcal{T}$;
2. each query Q contains two atoms and thus, by Lemma 5, the application of $\mathsf{UnfoldDB}$ to each Q is polynomial in the size of the mapping $\mathcal{M}$ and constant in the size of the data sources;
3. the evaluation of a union of SQL queries over a database can be computed in LogSpace with respect to the size of the database (since we assume that the SQL queries belong to the class of first-order logic queries). □

Theorem 7. *Given a DL-Lite*$_\mathcal{A}$ *ontology with mappings* $\mathcal{O}_m$*, and a UCQ* Q *over* $\mathcal{O}_m$*,* $\mathsf{Answer}(Q, \mathcal{O}_m)$ *runs* LogSpace *in the size of* $\mathcal{DB}$ *(data complexity). Moreover, it runs in polynomial time in the size of* $\mathcal{M}$*, in exponential time in the size of* Q*, and in polynomial time in the size of* $\mathcal{T}$*.*

Proof. The claim is a consequence of the results discussed in Section 2.4, i.e., the fact that $ans(Q, \mathcal{O}_m) = ans(\mathsf{PerfectRef}(Q, \mathcal{T}), db(\mathcal{A}(\mathcal{M}, \mathcal{DB})))$:

1. the maximum number of atoms in the body of a conjunctive query generated by the Algorithm $\mathsf{PerfectRef}$ is equal to the length of the initial query Q;
2. by Lemma 5, the algorithm $\mathsf{PerfectRef}$ $(Q, \mathcal{T})$ runs in time polynomial in the size of $\mathcal{T}$;
3. by Lemma 5, the cost of applying $\mathsf{UnfoldDB}$ to each conjunctive query in the union generated by $\mathsf{PerfectRef}$ has cost exponential in the size of the conjunctive query and polynomial in the size of $\mathcal{M}$; which implies that the query to be evaluated over the data sources can be computed in time exponential in the size of Q, polynomial in the size of $\mathcal{M}$ and constant in the size of $\mathcal{DB}$ (data complexity);
4. the evaluation of a union of SQL queries over a database can be computed in LogSpace with respect to the size of the database. □

7 Conclusions

We have studied the issue of ontology-based data access, under the fundamental assumption of keeping the data sources and the conceptual layer of an information system separate and independent. The solution provided in this paper

is based on the adoption of the *DL-Lite*$_{\mathcal{A}}$ description logic, which distinguishes between objects and values, and allows for connectingto external databases via suitable mappings. Notably, such a description logic admits advanced forms of reasoning, including satisfiability and query answering (with incomplete information), that are LogSpace in the size of the data at the sources. Even more significant from a practical point of view, *DL-Lite*$_{\mathcal{A}}$ allows for reformulating such forms of reasoning in terms of the evaluation of suitable SQL queries issued over over the sources, while taking into account and solving the impedance mismatch between data and objects.

We are currently implementing our solution on top of the QuOnto system[14] [2], a tool for reasoning over ontologies of the *DL-Lite* family. QuOnto was originally based on *DL-Lite*$_{\mathcal{F}}$, a DL that does not distinguish between data and objects. By enhancing QuOnto with the ability of reasoning both over *DL-Lite*$_{\mathcal{A}}$ ontologies and mappings, we have obtained a complete system for ontology-based data access.

While the possibility of reducing reasoning to query evaluation over the sources can only be achieved with description logics that are specifically tailored for this, such as *DL-Lite*$_{\mathcal{A}}$, we believe that the ideas presented in this paper on how to map a data layer to a conceptual layer and how to solve the impedance mismatch problem are of general value and can be applied to virtually all ontology formalisms.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publ. Co., 1995.
2. Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QuOnto: QUerying ONTOlogies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, pages 1670–1671, 2005.
3. Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 364–369, 2005.
4. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
5. Jesus Barrasa, Oscar Corcho, and Asuncion Gomez-Perez. R2O, an extensible and semantically based database-to-ontology mapping language. In *Proc. of the 7th Int. Workshop on the Web and Databases (WebDB 2004)*, 2004.
6. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. DL-Lite: Tractable description logics for ontologies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, pages 602–607, 2005.
7. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. In *Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 260–270, 2006.

[14] `http://www.dis.uniroma1.it/~quonto/`

8. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. of Automated Reasoning*, 2007. To appear.
9. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Data integration in data warehousing. *Int. J. of Cooperative Information Systems*, 10(3):237–271, 2001.
10. Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Deduction in concept languages: From subsumption to instance checking. *J. of Logic and Computation*, 4(4):423–452, 1994.
11. Francois Goasdoue, Veronique Lattes, and Marie-Christine Rousset. The use of CARIN language and algorithms for information integration: The Picsel system. *Int. J. of Cooperative Information Systems*, 9(4):383–401, 2000.
12. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the 12th Int. World Wide Web Conf. (WWW 2003)*, pages 48–57, 2003.
13. Jeff Heflin and James Hendler. A portrait of the Semantic Web in action. *IEEE Intelligent Systems*, 16(2):54–59, 2001.
14. Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.
15. Richard Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press, 1988.
16. Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. of the 16th Int. Conf. on Very Large Data Bases (VLDB'90)*, pages 455–468, 1990.
17. Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Data complexity of reasoning in very expressive description logics. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 466–471, 2005.
18. H. J. Komorowski. A specification of an abstract Prolog machine and its application to partial evaluation. Technical Report LSST 69, Linköping University, 1981.
19. Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002)*, pages 233–246, 2002.
20. John W. Lloyd. *Foundations of Logic Programming (Second, Extended Edition)*. Springer, Berlin, Heidelberg, 1987.
21. John W. Lloyd and John .C. Shepherdson. Partial evaluation in logic programming. *J. of Logic Programming*, 11:217–242, 1991.
22. Carsten Lutz. Description logics with concrete domains: A survey. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advances in Modal Logics*, volume 4. King's College Publications, 2003.
23. Alexander Mädche, Boris Motik, Nuno Silva, and Raphael Volz. MAFRA – a mapping framework for distributed ontologies. In *Proc. of the 13th Int. Conf. on Knowledge Engineering and Knowledge Management – Ontologies and the Semantic Web (EKAW 2002)*, pages 235–250, 2002.
24. José Meseguer and Xiaolei Qian. A logical semantics for object-oriented databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 89–98, 1993.
25. Jack Minker. A logic-based approach to data integration. *Theory and Practice of Logic Programming*, 2(3):293–321, 2002.

26. Maria Magdalena Ortiz, Diego Calvanese, and Thomas Eiter. Characterizing data complexity for conjunctive query answering in expressive description logics. In *Proc. of the 21st Nat. Conf. on Artificial Intelligence (AAAI 2006)*, 2006.
27. Antonella Poggi. *Structured and Semi-Structured Data Integration.* PhD thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 2006.
28. Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Databases*, pages 119–140. Plenum Publ. Co., 1978.
29. Francois Scharffe and Jos de Bruijn. A language to specify mappings between ontologies. In *Proc. of the 1st Int. Conf. on Signal-Image Technology and Internet-Based Systems (SITIS 2005)*, pages 267–271, 2005.
30. Luciano Serafini and Andrei Tamilin. DRAGO: Distributed reasoning architecture for the Semantic Web. In *Proc. of the 2nd European Semantic Web Conf. (ESWC 2005)*, volume 3532 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 2005.
31. Moshe Y. Vardi. The complexity of relational query languages. In *Proc. of the 14th ACM SIGACT Symp. on Theory of Computing (STOC'82)*, pages 137–146, 1982.