http://webdam.inria.fr/

Web Data Management

Ontologies

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

Contents

1 Introduction **2**

2 Ontologies by example **4**

3 RDF, RDFS, and OWL **7**
3.1 Web resources, URI, namespaces 7
3.2 RDF 8
3.3 RDFS: RDF Schema 10
3.4 OWL 13

4 Ontologies and (Description) Logics **17**
4.1 Preliminaries: the DL jargon 18
4.2 $\mathcal{ALC}$: the prototypical DL 21
4.3 Simple DLs for which reasoning is polynomial 24
4.4 The DL-LITE family: a good trade-off 26

5 Further reading **27**

6 Exercises **28**
.

1 Introduction

The vision of the Semantic Web is that of a world-wide distributed architecture where data and services easily interoperate. This vision is not yet a reality in the Web of today, in which given a particular need, it is difficult to find a resource that is appropriate to it. Also, given a relevant resource, it is not easy to understand what it provides and how to use it. To solve such limitations, facilitate interoperability, and thereby enable the Semantic Web vision, the key idea is to also publish *semantics descriptions* of Web resources. These descriptions rely on *semantic annotations*, typically on logical assertions that relate resources to some terms in predefined *ontologies*. This is the topic of the chapter.

An ontology is a formal description providing human users a shared understanding of a given domain. The ontologies we consider here can also be interpreted and processed by machines thanks to a logical semantics that enables reasoning. Ontologies provide the basis for sharing knowledge and as such, they are very useful for a number of reasons:

Organizing data. It is very easy to get lost in large collections of documents. An ontology is a natural means of "organizing" (structuring) it and thereby facilitates browsing through it to find interesting information. It provides an organization that is flexible, and that naturally structures the information in multidimensional ways. For instance, an ontology may allow browsing through the courses offered by a university by topic or department, by quarter or time, by level, etc.

Improving search. Ontologies are also useful for improving the accuracy of Web search. Consider a typical keyword search, say "jaguar USA". The result is a set of pages in which these intrinsically ambiguous English terms occur. Suppose instead that we use precise *concepts* in an ontology, say `car:jaguar country:USA`. First, one doesn't miss pages where synonyms are used instead of the query terms, e.g., United States. Also, one doesn't recover pages where one of the terms is used with a different meaning, e.g., pages that talk about the jaguar animal.

Data integration. Ontologies also serve as semantic glue between heterogeneous information sources, e.g., sources using different terminologies or languages. Consider for instance a French-American university program. The American data source will speak of "students" and "course", whereas the French one will use "étudiants" and "cours". By *aligning* their ontologies, one can integrate the two sources and offer a unique bilingual entry point to the information they provide.

An essential aspect of ontologies is their potential, because of the "logic inside", to be the core of *inferencing* components. What do we mean by inferencing in our setting? Consider for instance a query that is posed to the system. It may be the case that the query has no answer. It is then useful to infer why this is the case, to be able, for instance, to propose a more general query that will have some answers. On the other hand, the query may be too vague and have too many answers and it may be helpful to propose more specific queries that will help the user to precise what he really wants. In general, automatic inferences, even very simple ones, can provide enormous value to support user navigation and search, by guiding in a possibly overwhelming ocean of information. Inferencing is also an essential ingredient for automatically integrating different data sources. For instance, it is typically used to detect inconsistencies between data sources and resolve them, or to analyze redundancies and optimize query evaluation.

The inferencing potential of ontologies is based on their logical formal semantics. As we will see, languages for describing ontologies can be seen as fragments of first-order logic (FOL). Since inference in FOL is in general undecidable, the "game" consists in isolating fragments of FOL that are large enough to describe the semantics of resources of interest for a particular application, but limited enough so that inference is decidable, and even more, feasible in reasonable time.

Not surprisingly, we focus here on Web languages. More precisely, we consider languages that are already standards of the W3C or on the way to possibly becoming such standards (i.e., recommendations of that consortium). Indeed, in the first part of this chapter, we consider RDF, RDFS and OWL. Statements in these languages can be interpreted with the classical model-theoretic semantics of first-order logic.

In the second part, we study more formally, the inference problem for these languages. Checking logical entailment between formulas, possibly given a set of axioms, has been extensively studied. Since the problem is undecidable for FOL, we focus on decidable fragments of FOL that are known under the name of *description logics*. Description logics provide the formal basis of the OWL language recommended by the W3C for describing ontologies. They allow expressing and reasoning on complex logical axioms over unary and binary predicates. Their computational complexity varies depending on the set of constructors allowed in the language. The study of the impact of the choice of constructors on the complexity of inference is the main focus of the second part of the chapter.

We start with an example for illustrating what an ontology is, and the kind of reasoning that can be performed on it (and possibly on data described using it). Then, we survey the RDF(S) and OWL languages. Finally, we relate those languages to FOL, and in particular to description logics, in order to explain how the constructors used to describe an ontology may impact the decidability and tractability of reasoning on it.

2 Ontologies by example

An ontology is a formal description of a domain of interest based on a set of *individuals* (also called entities or objects), *classes* of individuals, and the *relationships* existing between these individuals. The logical statements on memberships of individuals in classes or relationships between individuals form a base of *facts*, i.e., a *database*. Besides, logical statements are used for specifying knowledge about the classes and relationships. They specify constraints on the database and form the *knowledge base*. When we speak of ontology, one sometimes thinks only of this knowledge that specify the domain of interest. Sometimes, one includes both the facts and the constraints under the term ontology.

In this chapter, we use as running example, a university ontology. In the example, the terms of the ontology are prefixed with ":", e.g., the individual :`Dupond` or the class :`Students`. This notation will be explained when we discuss name spaces.

The university ontology includes classes, e.g., :`Staff`, :`Students`, :`Department` or :`Course`. These classes denote natural *concepts* that are shared or at least understood by users familiar with universities all over the world. A class has a set of *instances* (the individuals in this class). For example, :`Dupond` is an instance of the class :`Professor`. The ontology also includes relationships between classes, that denote natural relationships between individuals in the real world. For instance, the university ontology includes the relationship, e.g., :`TeachesIn`. Relationships also have instances, e.g., `TeachesIn`(:`Dupond`,:`CS101`), is an instance of :`TeachesIn` that has the meaning that Dupond teaches CS101. Class or relationship instances form the database.

Let us now turn to the knowledge base. Perhaps the most fundamental constraint considered in this context is the `subclass` relationship. A class `C` is a subclass of a class `C'` if each instance of C is also an instance of C'. In other words, the set of instances of `C` is a subset of the set of instances of `C'`. For instance, by stating that the class :`Professor` is a subclass of the class :`AcademicStaff`, one expresses a knowledge that is shared with the university setting: all professors are members of the academic staff. Stating a subclass relationship between the class :`AcademicStaff` and the class :`Staff` expresses that all the members of the academic staff, in particular the professors, belong to the staff of the university. So, in particular, from the fact that :`Dupond` is an instance of the class :`Professor`, we also know that he is an instance of :`AcademicStaff` and of :`Staff`.

It is usual to represent the set of subclass statements in a graphical way by a *class hierarchy* (also called a *taxonomy*). Figure 1 shows a class hierarchy for the university domain.

Besides the class hierarchy, a very important class of ontology constraints allows fixing the domains of relationships. For instance,

- :`TeachesIn`(:`AcademicStaff`, :`Course`) indicates that if one states that "X :`TeachesIn` Y", then X belongs to :`AcademicStaff` and Y to :`Course`,
- Similarly, :`TeachesTo`(:`AcademicStaff`, :`Student`), :`Leads`(:`Staff`, :`Department`) indicate the nature of participants in different relationships.

A wide variety of other useful constraints are supported by ontology languages. For instance:

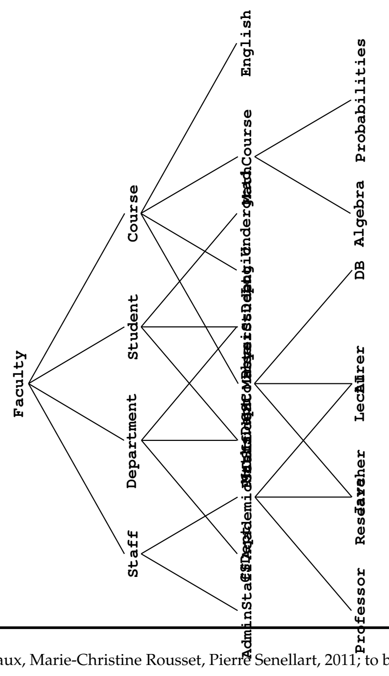

Figure 1: A class hierarchy

- Disjointness constraints between classes such as the classes `:Student` and `:Staff` are disjoint, i.e., a student cannot belong to the staff.
- Key constraints (for binary relations) such as each department must have a unique manager
- Domain constraints such as only professors or lecturers may teach undergraduate courses.

We will show how to give a precise (formal) semantics to these different kinds of constraints based on logic. The use of logic enables *reasoning*. For instance, from the fact that Dupond leads the CS department and the university ontology, it can be logically inferred that `:Dupond` is in `:Professor` and `:CSDept` is in `:Department`. Indeed, such a reasoning based on ontologies and *inference rules* is one of the main topics of this chapter. But before delving in technicalities on inference, we devote the remaining of the section to illustrations of the usefulness of inference.

Inference is first very useful for query answering. For instance, consider the query "Who are the members of the academic staff living in Paris?". Suppose Professor `:Dupond` lives in Paris. He should be in the answer. Why? Because he lives in Paris and because he is a professor. Note however that the only explicit facts we may have for `:Dupond` are that he lives in Paris and that he is a professor. A query engine will have to use the formula that states that professors are members of the academic staff and *inference* to obtain Dupond in the answer. Without inference, we would miss the result.

Inference helped us in the previous example derive new facts such as Dupond is member of the academic staff. It can also serve to derive new knowledge, i.e., new constraints even in absence of any fact. For instance, suppose that we add to the class hierarchy of Figure 1, the subclass relationship between `:PhDStudent` and `:Lecturer`. Then, it can be inferred that `:PhDStudent` is also a subclass of the class `:Staff`. At the time one is designing such an ontology, it is useful to be aware of such inference. For instance, membership in the staff class may bring special parking privileges. Do we really mean to give such privileges to all PhD students?

Furthermore, suppose that the ontology specifies the (already mentioned) disjointness relationship between the classes `:Staff` and `:Student`. Then, from this constraint and the subclass relationship between `:PhDStudent` and `:Student`, it can be inferred that the class `:PhDStudent` is empty. This should be understood as an anomaly: why would a specification bother to define an empty class? Highlighting in advance such anomalies in ontologies is very important at the time the ontology is defined, because this may prevent serious errors at the time the ontology is actually used in an application.

A last illustration of the use of ontologies pertains to integration. Consider again an international university program. Suppose US students may follow some courses in France for credits. Then we need to integrate the French courses with the American ones. Statement such as "FrenchUniv`:Cours` is a subclass of `:Course`" serves to map French concepts to American ones. Now a student in this international program who would ask the query "database undergraduate", may get as answers *Database 301* and *L3, Bases de données*.

These are just examples to illustrate the usefulness of inference based on ontologies. In the next section we describe the languages and formalisms that can be used to represent ontologies. In Section 4, we will be concerned with inference algorithms that are sound and complete with respect to the logical formal semantics, that is, algorithms guaranteeing to infer all the implicit information (data or knowledge) deriving from the asserted facts, relationships and constraints declared in the ontology.

3 RDF, RDFS, and OWL

We focus on three ontology languages that have been proposed for describing Web resources. We first consider the language RDF, a language for expressing facts (focusing primarily on the database). The other two languages allow constraining RDF facts in particular application domains: RDFS is quite simple, whereas OWL is much richer. We start by reviewing common terminology and notions central to this context.

3.1 Web resources, URI, namespaces

A *resource* is anything that can be referred to: a Web page, a fragment of an XML document (identified by an element node of the document), a Web service, an identifier for an entity, a thing, a concept, a property, etc. This is on purpose very broad. We want to be able to talk about and describe (if desired) anything we can identify. An URI may notably be a URL that any (human or software) agent or application can access. In particular, we need to talk about specific ontologies. The ontology that is used in the example of this chapter is identified by the URL:

```
http://Webdam.inria.fr/Jorge/OntologiesChapter/Examples#
```

The instance Dupond in this ontology has URI:

```
http://Webdam.inria.fr/Jorge/OntologiesChapter/Examples#Dupond
```

To avoid carrying such long URIs, just like in XML (see Chapter **??**), we can use namespaces. So for instance, we can define the *namespace* `jorge`: with the URL of the example ontology. Then the instance Dupond in the ontology `jorge`: becomes `jorge`:`Dupond`. This is just an abbreviation of the actual URI for Dupond in that ontology.

The examples we will present are within the `jorge` ontology. When denoting individuals or relationships in this ontology, we will use the notation :`Name` instead of `jorge`:`Name`, considering that `jorge` is the default namespace. For example, the RDF triplet ⟨ :`Dupond`, :`Leads`, :`CSDept` ⟩ expresses in RDF the fact that :`Dupond` leads :`CSDept`. Remember that these are only abbreviations, e.g.,:
`http://Webdam.inria.fr/Jorge/OntologiesChapter/Examples#Dupond`
abbreviates to
`Jorge`:`Dupond`
abbreviates to
:`Dupond`.

One can publish standard namespaces to be used by all those interested in particular domain areas. In this chapter, we will use the following standard namespaces:

`rdf:` A namespace for RDF.
The URI is: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

`rdfs:` A namespace for RDFS.
The URI is: `http://www.w3.org/2000/01/rdf-schema#`

owl: A namespace for OWL.
The URI is: `http://www.w3.org/2002/07/owl#`

dc: A namespace for the Dublin Core Initiative.
The URI is: `http://dublincore.org/documents/dcmi-namespace/`

foaf: A namespace for FOAF.
The URI is: `http://xmlns.com/foaf/0.1/`.

In each case, at the URL, one can find an ontology that, in particular, specifies a particular vocabulary. Dublin Core is a popular standard in the field of digital libraries. The Friend of a Friend (FOAF) initiative aims at creating a "social" Web of machine-readable pages describing people, the links between them and the things they create and do. We will encounter examples of both.

3.2 RDF

RDF (Resource Description Framework) provides a simple language for describing *annotations* about Web resources identified by URIs. These are facts. Constraints on these facts in particular domains will be stated in RDFS or OWL.

RDF syntax: RDF triplets.

In RDF, a fact expresses some metadata about a resource that is identified by a URI. An RDF fact consists of a *triplet*. A triplet is made of a *subject*, a *predicate* and an *object*. It expresses a *relationship* denoted by the *predicate* between the *subject* and the *object*. Intuitively, a triplet $\langle a\ P\ b \rangle$ expresses the fact that b is a value of *property* P for the subject a. (In general, a may have several values for property p.) Don't get confused by the terminology: a relationship, a predicate, a property, are three terms for the same notion. The relationship $\langle a\ P\ b \rangle$ uses the predicate P, and expresses that the subject a has value b for property P.

In a triplet, the subject, but also the predicate, are URIs pointing to Web resources, whereas the object may be either a URI or a *literal* representing a *value*. In the latter case, a triplet expresses that a given subject has a given value for a given property. RDF borrows from XML the literal data types such as strings, integers and so forth, thanks to the predefined RDF data type `rdf:Literal`. One can include an arbitrary XML value as an object of an RDF triplet, by using the predefined RDF data type `rdf:XMLLiteral`.

In RDF, one can distinguish between individuals (objects) and properties (relationships). This is not mandatory but it can be done using two `rdf` keywords (i.e., keywords defined in the `rdf` namespace): `rdf:type` and `rdf:Property`. For instance, one can declare that the term `:Leads` is a property name by the triplet ⟨ `:Leads rdf:type rdf:Property` ⟩.

Then data is specified using a set of triplets. These triplets may be represented either in a tabular way, as a *triplet table* or as a *RDF graph*.

Representing a set of triplets as a directed *graph* is convenient to visualize all the information related to an individual at a single node by bringing it together. In such a graph, each triplet is represented as an edge from its subject to its object. Figure 2 and Figure 3 visualize respectively the tabular form and the RDF graph corresponding to some set of triplets:

This is almost all there is in RDF. Trivial, no? There is one last feature, the use of blank nodes to capture some form of unknown individuals. A *blank node* (or anonymous resource

⟨ :Dupond :Leads :CSDept ⟩
⟨ :Dupond :TeachesIn :UE111 ⟩
⟨ :Dupond :TeachesTo :Pierre ⟩
⟨ :Pierre :EnrolledIn :CSDept ⟩
⟨ :Pierre :RegisteredTo :UE111 ⟩
⟨ :UE111 :OfferedBy :CSDept ⟩

Subject	**Predicate**	**Object**
:Dupond	:Leads	:CSDept
:Dupond	:TeachesIn	:UE111
:Dupond	:TeachesTo	:Pierre
:Pierre	:EnrolledIn	:CSDept
:Pierre	:RegisteredTo	:UE111
:UE111	:OfferedBy	:CSDept

Figure 2: An RDF triplet table

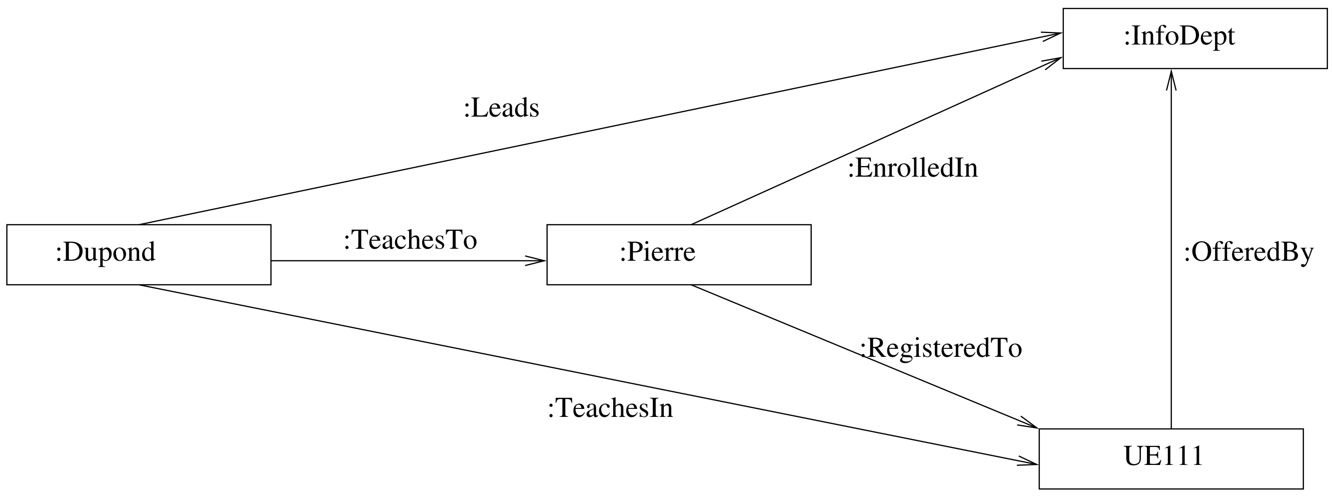

Figure 3: An RDF graph

or bnode) is a subject or an object in an RDF triplet or an RDF graph that is not identified by a URI and is not a literal. A blank node is referred to by a notation _:p where p is a local name that can be used in several triplets for stating several properties of the corresponding blank node.

Example 3.1 *The following triplets express that Pierre knows someone named "John Smith" wrote a book entitled "Introduction to Java".*

:Pierre	*foaf:knows*	*_:p*
_:p	*foaf:name*	*"John Smith"*
_:p	*wrote*	*_:b*
_:b	*dc:title*	*"Introduction to Java"*

The predicates foaf:knows and foaf:name belong to the FOAF vocabulary. The predicate dc:title belongs to the Dublin Core vocabulary.

We have used here an abstract syntax of RDF. One can clearly describe in a number of ways using a "concrete" syntax. For instance, there exists an RDF/XML syntax for describing RDF triplets. We now turn to the semantics that is, as we will see, quite simple as well.

RDF semantics

A triplet $\langle s \;\; P \;\; o \rangle$ without blank node is interpreted in first-order logic (FOL) as a fact $P(s,o)$, i.e., a grounded atomic formula, where P is the name of a predicate and s and o denotes constants in the FOL language.

Blank nodes, when they are in place of the *subject* or the *object* in triplets, are interpreted as *existential variables*. Therefore a set of RDF triplets (represented with a triplet table or an RDF graph or in RDF/XML syntax), possibly with blank nodes as subjects or objects, is interpreted as a conjunction of positive literals in which all the variables are existentially quantified.

Giving a FOL semantics to triplets in which the predicates can be blank nodes is also possible but a little bit tricky and is left out of the scope of this chapter (see Section 5).

Example 3.2 *Consider again the four triplets that we used to express that Pierre knows someone named "John Smith" wrote a book entitled "Introduction to Java". They are interpreted in FOL by the following* positive existential conjunctive formula, *where the prefixes (*`foaf:`, `dc:`, `_:` *and* `:`*) for denoting the constants, predicates and variables have been omitted for readability.*

$$\exists p \exists b [knows(Pierre, p) \wedge name(p, \text{"John Smith"}) \wedge wrote(p, b) \wedge title(b, \text{"Introduction to Java"})]$$

3.3 RDFS: RDF Schema

RDFS is the *schema language* for RDF. It allows specifying a number of useful constraints on the individuals and relationships used in RDF triplets. In particular, it allows declaring objects and subjects as *instances* of certain *classes*. In addition, *inclusion statements* between classes and properties make it possible to express semantic relations between classes and between properties. Finally, it is also possible to semantically relate the "domain" and the "range" of a property to some classes. These are all very natural constraints that we consider next.

Syntax of RDFS

The RDFS statements can be themselves expressed as RDF triplets using some specific predicates and objects used as RDFS keywords with a particular meaning. We have already seen the `rdf:type` predicate. This same keyword is used to declare that an individual `i` is an instance of class `C` with a triplet of the form ⟨`i rdf:type C` ⟩.

Example 3.3 *The following triplets express that* `:Dupond` *is an instance of the class* `:AcademicStaff`, `:UE111` *of the class* `:Java` *and* `:Pierre` *is an instance of the class* `:MasterStudent`.

```
:Dupond  rdf:type :AcademicStaff
:UE111   rdf:type :Java
:Pierre  rdf:type :MasterStudent
```

RDFS provides a new predicate `rdfs:subClassOf` to specify that a class is a subclass of another one. One can use it in particular to disambiguate terms. For instance, by specifying that `:Java` is a subclass of `:CSCourse`, one say that, in the context of this particular ontology, by Java, we mean exclusively the CS programming language. Subclass relationships between classes, and thus a class hierarchy, can be declared as a set of triplets of the form ⟨ C `rdfs:subClassOf` D ⟩. The class hierarchy of Figure1 can be described in RDFS by a set of RDF triplets, an extract of which is given in Figure 4.

```
:Java           rdfs:subClassOf  :CSCourse
:AI             rdfs:subClassOf  :CSCourse
:BD             rdfs:subClassOf  :CSCourse
:CSCourse       rdfs:subClassOf  :Course
:Logic          rdfs:subClassOf  :Course
:MathCourse     rdfs:subClassOf  :Course
:English        rdfs:subClassOf  :Course
:Algebra        rdfs:subClassOf  :MathCourse
:Probabilities  rdfs:subClassOf  :MathCourse
```

Figure 4: Some RDFS declarations for the class hierarchy of Figure 1

Similarly, RDFS provides a predicate `rdfs:subPropertyOf` to express structural relationships between properties. We could state for instance with the triplet

$$\langle \texttt{:LateRegisteredTo rdfs:subPropertyOf :RegisteredTo} \rangle$$

that the relationship `:LateRegisteredTo` is more specific than the relationship `:RegisteredTo`. So, suppose that we know:

$$\langle \texttt{:Alice :LateRegisteredTo :UE111} \rangle$$

Then we can infer that, also:

$$\langle \texttt{:Alice :RegisteredTo :UE111} \rangle$$

A property P (between subjects and objects) may be seen as a function that maps a subject s to the set of objects related to s via P. This functional view motivates calling the set of subjects of a property P, its *domain*, and the set of objects, its *range*.

Restricting the domain and the range of a property is also possible in RDFS using two other new predicates `rdfs:domain` and `rdfs:range` and triplets of the form:

$$\langle P \texttt{ rdfs:domain } C \rangle$$

and

$$\langle P \texttt{ rdfs:range } D \rangle$$

Example 3.4 *Some domain and range constraints on properties in the ontology of the university domain mentioned in Section 2 can be expressed in RDFS by the set of RDF triplets given in Figure 5.*

:TeachesIn	rdfs:domain	:AcademicStaff	:TeachesIn	rdfs:range	:Course
:TeachesTo	rdfs:domain	:AcademicStaff	:TeachesTo	rdfs:range	:Student
:Leads	rdfs:domain	:Staff	:Leads	rdfs:range	:Department

Figure 5: Some RDFS declarations of domain and range constraints for the university domain

RDFS semantics

Accordingly to the FOL semantics of RDF presented before, the RDFS statements can be interpreted by FOL formulas. Figure 6 gives the logical semantics of the RDFS statements by giving their corresponding FOL translation. The figure also gives the corresponding DL notation, to be explained further on.

RDF and RDFS statements	**FOL translation**	**DL notation**
`i rdf:type C`	$C(i)$	$i : C$ or $C(i)$
`i P j`	$P(i,j)$	$i\,P\,j$ or $P(i,j)$
`C rdfs:subClassOf D`	$\forall X\,(C(X) \Rightarrow D(X))$	$C \sqsubseteq D$
`P rdfs:subPropertyOf R`	$\forall X \forall Y\,(P(X,Y) \Rightarrow R(X,Y))$	$P \sqsubseteq R$
`P rdfs:domain C`	$\forall X \forall Y\,(P(X,Y) \Rightarrow C(X))$	$\exists P \sqsubseteq C$
`P rdfs:range D`	$\forall X \forall Y\,(P(X,Y) \Rightarrow D(Y))$	$\exists P^{-} \sqsubseteq D$

Figure 6: RDFS logical semantics

Observe that these statements all have the same general form:

$$\forall \ldots (\cdots \Rightarrow \ldots)$$

Such constraints are very useful in practice and are very adapted to inferencing. They are called *tuple generating dependencies*. Intuitively, each such rule may be thought of as a factory for generating new facts: no matter how (with which valuations of the variables) you can match the left part of the arrow, you can derive the right part. Underneath this inference are the notion of *pattern*, i.e., of a fact where some of the individuals are replaced by variables (i.e., placeholder) and that of *valuation*. An example of a pattern is `:TeachesIn`(X, Y), where X, Y are variables. A valuation ν may map X to `:Dupond` and Y to `:UE111`. It transforms the pattern `:TeachesIn`(X, Y) into the fact `:TeachesIn`(`:Dupond`, `:UE111`).

The FOL translation that we presented suggests inference rules that can be used "operationally" to derive new RDF triplets. This is what is called the *operational semantics* of RDFS. One starts with a set of facts, RDF triplets, and constraints. When the body of a rule matches some knowledge we have, the head of the rule specifies some knowledge we can infer. To illustrate, consider the inference rule for `rdfs:subClassOf`:

if $\langle$ r `rdf:type` A $\rangle$ and $\langle$ A `rdfs:subClassOf` B $\rangle$ **then** $\langle$ r `rdf:type` B $\rangle$

where r, A, and B are variables.

This means that: **if** we know two triplets matching the patterns $\langle$ r `rdf:type` A $\rangle$ and $\langle$ A `rdfs:subClassOf` B $\rangle$ for some values of r, A, B, **then** we can infer the triplet $\langle$ r `rdf:type` B $\rangle$ with the values of r, B taken to be those of the match.

Or more formally, if there exists a valuation ν such that we know:

$$\langle\ \nu(r)\ \texttt{rdf:type}\ \nu(A)\ \rangle$$

and

$$\langle\ \nu(A)\ \texttt{rdfs:subClassOf}\ \nu(B)\ \rangle$$

then we can infer:

$$\langle\ \nu(r)\ \texttt{rdf:type}\ \nu(B)\ \rangle$$

Of course, triplets that have been inferred can be themselves used to infer more triplets.

We also need rules to capture the operational semantics of `rdfs:subPropertyOf`, `rdfs:domain` and `rdf:range` constraints:

if $\langle\ r\ P\ s\ \rangle$ **and** $\langle\ P\ \texttt{rdfs:subPropertyOf}\ Q\ \rangle$ **then** $\langle\ r\ Q\ s\ \rangle$.
if $\langle\ P\ \texttt{rdfs:domain}\ C\ \rangle$ **and** $\langle\ x\ P\ y\ \rangle$ **then** $\langle\ x\ \texttt{rdf:type}\ C\ \rangle$.
if $\langle\ P\ \texttt{rdfs:range}\ D\ \rangle$ **and** $\langle\ x\ P\ y\ \rangle$ **then** $\langle\ \mathrm{y}\ \texttt{rdf:type}\ D\ \rangle$.

An important issue is that of the *soundness* and *completeness* of the operational semantics defined by the four above inference rules. Let $\mathcal{A}$ be a set of RDF triplets and $\mathcal{T}$ be a set of associated RDFS triplets (expressing constraints on facts in $\mathcal{A}$). The operational semantics is *sound* if any fact f inferred from $\mathcal{A}$ and $\mathcal{T}$ by the rules (denoted: $\mathcal{A} \cup \mathcal{T} \vdash f$) is a logical consequence of the facts in $\mathcal{A}$ together with the constraints in $\mathcal{T}$ (denoted: $\mathcal{A} \cup \mathcal{T} \models f$).

The soundness of the operational semantics is easy to show, just because our rules are very close to the constraints imposed by the RDFS statements. More formally it can be shown by induction on the number of rules required to infer f (details are left as an exercise).

It is a little bit more difficult to show that the operational semantics is *complete*, i.e., for any fact f, if $\mathcal{A} \cup \mathcal{T} \models f$ then $\mathcal{A} \cup \mathcal{T} \vdash f$. We prove it by contrapositive, i.e., we show that if $\mathcal{A} \cup \mathcal{T} \nvdash f$ then $\mathcal{A} \cup \mathcal{T} \not\models f$. We consider the set of constants appearing in the facts in $\mathcal{A}$ as the domain of an interpretation I, which is built as follows from the set of facts obtained by applying the rules to the set of triplets in $\mathcal{A} \cup \mathcal{T}$:

- for each class C, $I(C) = \{i | \mathcal{A} \cup \mathcal{T} \vdash C(i)\}$
- for each property R, $I(R) = \{(i,j) | \mathcal{A} \cup \mathcal{T} \vdash R(i,j)\}$

It is easy to verify that I is a model of $\mathcal{A} \cup \mathcal{T}$ (details are left as an exercise). Now, let f be a fact such that $\mathcal{A} \cup \mathcal{T} \nvdash f$. By construction of I, since f is not inferred by the rules, f is not true in I. Therefore, I is a model of $\mathcal{A} \cup \mathcal{T}$ in which f is not true, i.e., $\mathcal{A} \cup \mathcal{T} \not\models f$.

It is important to note that the completeness we just discussed concerns the inference of *facts* (possibly with blank nodes). For extending the completeness result to the inference of *constraints*, we need additional inference rules (described in Figure 7) to account for the combination of range and domain constraints with the subclass relationship, and also for expressing the transitivity of the subclass and subproperty constraints. The proof is left as an exercise (see Exercise 6.1).

The RDFS statements are exploited to saturate the RDF triplets by adding the triplets that can be inferred with the rules. Then the resulting set of RDF triplets can be queried with a query language for RDF, e.g., SPARQL. This will be explained in the next chapter.

3.4 OWL

OWL (the Web Ontology Language and surprisingly not the Ontology Web Language) extends RDFS with the possibility to express additional constraints. Like RDFS, OWL statements can be expressed as RDF triplets using some specific predicates and objects used as OWL keywords with a particular meaning. In this section, we describe the main OWL constructs.

if $\langle$ P `rdfs:domain` A $\rangle$ **and** $\langle$ A `rdfs:subClassOf` B $\rangle$
  then $\langle$ P `rdfs:domain` B $\rangle$
if $\langle$ P `rdfs:range` C $\rangle$ **and** $\langle$ C `rdfs:subClassOf` D $\rangle$
  then $\langle$ P `rdfs:range` D $\rangle$
if $\langle$ A `rdfs:subClassOf` B $\rangle$ **and** $\langle$ B `rdfs:subClassOf` C $\rangle$
  then $\langle$ A `rdfs:subClassOf` C $\rangle$
if $\langle$ P `rdfs:subPropertyOf` Q $\rangle$ **and** $\langle$ Q `rdfs:subPropertyOf` R $\rangle$
  then $\langle$ P `rdfs:subPropertyOf` R $\rangle$

Figure 7: The inference rules for RDFS constraints

Like RDFS, we provide their FOL semantics and, in anticipation to the next section, the corresponding DL notation.

There are many constructs expressible in OWL that provide considerable modeling flexibility and expressiveness for the Semantic Web. Providing an operational semantics for all the OWL constructs is an open issue. However, most of the OWL constructs come from DL. Therefore, we get for free all the positive and negative known results about reasoning in DLs. This allows better understanding inferences when considering facts described with RDF triplets and constraints in OWL.

OWL offers a number of rich semantic constructs, namely class disjointness, functional constraint, intentional class definition, class union and intersection, etc.. We consider them in turn.

Expressing class disjointness constraints

OWL provides a predicate `owl:disjointWith` to express that two classes `C` and `D` are disjoint using the triplet: $\langle$ C `owl:disjointWith` D $\rangle$

Although very natural, this constraint cannot be expressed in RDFS. For instance, in our example, we can state the triplet: $\langle$ `:Student` `owl:disjointWith` `:Staff` $\rangle$.

The following table provides the logical semantics of this construct.

OWL notation	**FOL translation**	**DL notation**
`C owl:disjointWith D`	$\forall X\,(C(X) \Rightarrow \neg D(X))$	$C \sqsubseteq \neg D$

Observe the use of negation in the logical formulas. This is taking us out of tuple generating dependencies. Such rules are not used to produce new facts but for ruling out possible worlds as inconsistent with what we know of the domain of interest.

Functional constraints

In OWL, it is possible to state that a given relationship between A and B is in fact a *function* from A to B (resp. from B to A). One can also state that a property is the *inverse* of another, or that a property is *symmetric*. Observe the use of equality in the logical formulas.

OWL notation	**FOL translation**	**DL notation**
P `rdf:type` `owl:FunctionalProperty`	$\forall X \forall Y \forall Z$ $(P(X,Y) \wedge P(X,Z) \Rightarrow Y = Z)$	$(funct\, P)$ or $\exists P \sqsubseteq (\leq 1\, P)$
P `rdf:type` `owl:InverseFunctionalProperty`	$\forall X \forall Y \forall Z$ $(P(X,Y) \wedge P(Z,Y) \Rightarrow X = Z)$	$(funct\, P^-)$ or $\exists P^- \sqsubseteq (\leq 1\, P^-)$
P `owl:inverseOf` Q $\forall X \forall Y (P(X,Y) \Leftrightarrow Q(Y,X))$	$P \equiv Q^-$	
P `rdf:type` `owl:SymmetricProperty`	$\forall X \forall Y (P(X,Y) \Rightarrow P(Y,X))$	$P \sqsubseteq P^-$

Recall that a triplet $\langle a\ P\ b \rangle$ is viewed in the model-theoretic interpretation as a pair in relation P. An `owl:FunctionalProperty` thus expresses that the first attribute of P is a *key*, while an `owl:InverseFunctionalProperty` expresses that its second attribute is a *key*. Note that a property may be both an `owl:FunctionalProperty` and an `owl:InverseFunctionalProperty`. It would be the case for instance for the property `hasIdentityNo` that associates identification numbers to students in the university example.

Example 3.5 *In the university example, the constraint that every department must be led by a unique manager is expressed in OWL by adding the triplet:*

`:Leads rdf:type owl:InverseFunctionalProperty`

to the RDFS triplets we already have for the domain and range constraints for `:Leads`. See Figure 5.

Intentional class definitions.

A main feature of OWL is the intentional definition of new classes from existing ones. It allows expressing complex constraints such as *every department has a unique manager who is a professor*, or *only professors or lecturers may teach to undergraduate students*.

The keyword `owl:Restriction` is used in association with a blank node class, that is being defined (without being given a name), and some specific restriction properties (`owl:someValuesFrom`, `owl:allValuesFrom`, `owl:minCardinality`, `owl:maxCardinality`) used for defining the new class. The blank node is necessary because the expression of each restriction requires a set of triplets that are all related to the same class description.

Example 3.6 *The following set of triplets defines the blank (i.e., unnamed) class describing the set of individuals for which* all *the values of the property `:Leads` come from the class `:Professor`:*

```
_:a  rdfs:subClassOf    owl:Restriction
_:a  owl:onProperty     :Leads
_:a  owl:allValuesFrom  :Professor
```

The constraint that every department can be led only *by professors is then simply expressed by adding the following triplet (involving the same blank class _:a):*

```
:Department  rdfs:subClassOf  _:a
```

Note that the constraint that *every department must be led by a unique manager who is a professor* is actually the conjunction of the above constraint and of the functionality constraint of `:Leads`.

Also with restriction, one can use the `owl:someValuesFrom` keyword on a property `P` to produce a class description denoting the set of individuals for which *at least one* value of the property `P` comes from a given class C using:

```
_:a  rdfs:subClassOf     owl:Restriction
_:a  owl:onProperty      P
_:a  owl:someValuesFrom  C
```

Finally, the `owl:minCardinality` and `owl:maxCardinality` restrictions allow expressing constraints on the number of individuals that can be related by a given property `P`.

Example 3.7 *The following triplets describe the class (denoted by the blank node _:a) of individuals having at least 3 registrations and the class (denoted by the blank node _:b) of individuals having atmost 6 registrations.*

```
_:a  rdfs:subClassOf     owl:Restriction
_:a  owl:onProperty      RegisteredTo
_:a  owl:minCardinality  3
_:b  rdfs:subClassOf     owl:Restriction
_:b  owl:onProperty      RegisteredTo
_:b  owl:maxCardinality  6
```

The constraint that each student must be registered to at least 3 courses and atmost 6 courses is then simply expressed by adding the two following triplets (involving the same blank classes _:a and _:b):

```
:Student  rdfs:subClassOf  _:a
:Student  rdfs:subClassOf  _:b
```

The logical semantics of these different class descriptions defined by restrictions can be given either in FOL as open formulas with one free variable or as DL concepts using DL constructors. This is summarized in Figure 8, where X denotes a free variable.

OWL notation	**FOL translation**	**DL notation**
_:a `owl:onProperty` P _:a `owl:allValuesFrom` C	$\forall Y\,(P(X,Y) \Rightarrow C(Y))$	$\forall P.C$
_:a `owl:onProperty` P _:a `owl:someValuesFrom` C	$\exists Y\,(P(X,Y) \wedge C(Y))$	$\exists P.C$
_:a `owl:onProperty` P _:a `owl:minCardinality` n	$\exists Y_1 \ldots \exists Y_n (P(X,Y_1) \wedge \ldots \wedge P(X,Y_n) \wedge \bigwedge_{i,j\in[1..n],i\neq j}(Y_i \neq Y_j))$	$(\geq n\,P)$
_:a `owl:maxCardinality` n	$\forall Y_1 \ldots \forall Y_n \forall Y_{n+1}$ $(P(X,Y_1) \wedge \ldots \wedge P(X,Y_n) \wedge P(X,Y_{n+1})$ $\Rightarrow \bigvee_{i,j\in[1..n+1],i\neq j}(Y_i = Y_j))$	$(\leq n\,P)$

Figure 8: Logical semantics of the OWL restriction constructs

Union and intersection.

The `owl:intersectionOf` and `owl:unionOf` constructs allow combining classes. The intersection of (possibly unnamed) classes denotes the individuals that belong to both classes; whereas the union denotes the individuals that belong to some. Note that the argument of those two constructs is in fact a *collection*, for which we use the standard shortcut notation of lists, as illustrated in the following example by the list (`:Professor, :Lecturer`) declared as the argument of `owl:unionOf`.

Example 3.8 *The constraint that* only professors or lecturers may teach to undergraduate students *can be expressed in OWL as follows:*

```
_:a  rdfs:subClassOf     owl:Restriction
_:a  owl:onProperty      :TeachesTo
_:a  owl:someValuesFrom  :Undergrad
_:b  owl:unionOf         (:Professor,:Lecturer)
_:a  rdfs:subClassOf     _:b
```

In the spirit of union, and like union requiring logical disjunction, the `owl:oneOf` construct allows describing a class by enumerating its elements as a collection. This corresponds to the following FOL and DL semantics.

OWL notation	**FOL translation**	**DL notation**
`owl:intersectionOf` $(C,D...)$	$C(X) \wedge D(X) \ldots$	$C \sqcap D \ldots$
`owl:unionOf` $(C,D...)$	$C(X) \vee D(X) \ldots$	$C \sqcup D \ldots$
`owl:oneOf` $(e,f...)$	$X = e \vee X = f \ldots$	`oneOf` $\{e,f,\ldots\}$

Class and property equivalence.

The construct `owl:equivalentClass` allows stating that two classes are equivalent, i.e., that there are inclusions both ways. Similarly, `owl:equivalentProperty` allows stating that two properties are equivalent. Strictly speaking, those two constructs do not add expressivity to RDFS. In fact, $\langle$ C `owl:equivalentClass` D $\rangle$ can be expressed in RDFS by the two triplets: $\langle$ C `rdfs:subClassOf` D $\rangle$ and $\langle$ D `rdfs:subClassOf` C $\rangle$.

As explained in the next section, OWL constructs are all syntactic variants of description logic constructors.

4 Ontologies and (Description) Logics

First-order logic (FOL) is the formal foundation of the OWL ontology Web language. First-order logic (also called predicate logic) is especially appropriate for knowledge representation and reasoning. In fact, ontologies are simply knowledge about classes and properties. From a logical point of view, classes are unary predicates while properties are binary predicates, and constraints are logical formulas asserted as axioms on these predicates, i.e., asserted as true in the domain of interest.

From the early days of computer science, the problem of automatic deduction in FOL has been extensively studied. The main result that any computer scientist should know is that *the implication problem in FOL is not decidable but only recursively enumerable*, which is stated briefly as *FOL is r.e.*. That means that there exists an algorithm that given some formula φ enumerates all the formulas ψ such φ implies ψ. On the other hand, there does not exist any general algorithm (i.e., a systematic machinery) that, applied to two any input FOL formulas φ and ψ, decides where φ implies ψ. Observe that φ implies ψ if and only if there is no model for $\varphi \wedge \neg\psi$. Thus the seemingly simpler problem of deciding whether a FOL formula is satisfiable is also not decidable.

A lot of research has then been devoted to exhibit *fragments* of FOL that are decidable, i.e., subsets of FOL formulas defined by some restrictions on the allowed formulas, for which checking logical entailment between formulas, possibly given a set of axioms, can be performed automatically by an algorithm. In particular, *description logics (DLs)* are decidable fragments of first-order logic allowing reasoning on complex logical axioms over unary and binary predicates. This is exactly what is needed for handling ontologies. Therefore, it is not surprising that the OWL constructs have been borrowed from DLs. DLs cover a broad spectrum of class-based logical languages for which reasoning is decidable with a computational complexity that depends on the set of constructs allowed in the language.

Research carried out on DLs provides a fine-grained analysis of the trade-off between expressive power and computational complexity of sound and complete reasoning. In this section, we just give a minimal background on the main DLs constructs and the impact of their combinations on the complexity of reasoning. This should first help practitioners to choose among the existing DL reasoners the one that is the most appropriate for their application. Also, for researchers it should facilitate further reading of advanced materials about DLs.

4.1 Preliminaries: the DL jargon

A DL knowledge base is made of an intentional part (the Tbox) and an assertional part (the Abox). The Tbox defines the ontology serving as conceptual view over the data in the Abox. In DLs, the classes are called *concepts* and the properties are called *roles*.

A Tbox $\mathcal{T}$ is a set of *terminological axioms* which state inclusions or equivalences between (possibly complex) concepts ($B \sqsubseteq C$ or $B \equiv C$) and roles ($R \sqsubseteq E$ or $R \equiv E$), while an Abox $\mathcal{A}$ is a set of *assertions* stating memberships of individuals in concepts ($C(a)$) and role memberships for pairs of individuals ($R(a,b)$). The legal DL knowledge bases $\langle \mathcal{T}, \mathcal{A} \rangle$ vary according to the DL *constructs* used for defining complex concepts and roles, and to the restrictions on the axioms that are allowed in the Tbox and the assertions allowed in the Abox. As said in the previous section, the DL *constructs* are the OWL *constructs*, denoted with a different syntax. The ingredients for constructing a DL knowledge base are thus:

- a *vocabulary* composed of a set **C** of *atomic concepts* (A, B...), a set **R** of *atomic roles* (P, Q...), and a set **O** of *individuals* (a, b, c...),
- a set of *constructs* used for building complex concepts and roles from atomic concepts and roles,
- a language of *axioms* that can be stated for constraining the vocabulary in order to express domain constraints.

Example 4.1 Student ⊓ Researcher *is a complex concept built from the two atomic concepts* Student *and* Researcher *using the* conjunction *construct (which is denoted `owl:intersectionOf` in OWL). This complex concept can be related to the atomic concept* PhDStudent *by an* inclusion axiom*:*

$$\text{PhDStudent} \sqsubseteq \text{Student} \sqcap \text{Researcher}$$

or by an equivalence axiom*:*

$$\text{PhDStudent} \equiv \text{Student} \sqcap \text{Researcher}$$

The difference between inclusion and equivalence axioms will be clearer when we will define the formal semantics underlying DLs. From a modeling point of view, the equivalence axioms are used to define new concepts (such as *PhDStudent*) from existing concepts (such as *Student* and *Researcher*). Concepts can be defined by restricting a role using either the *value restriction* construct $\forall R.C$ (denoted `owl:allValuesFrom` in OWL) or the *existential restriction* construct $\exists R.C$ (denoted `owl:someValuesFrom` in OWL). For example, we can define the concept *MathStudent* as follows:

$$MathStudent \equiv Student \sqcap \forall\ RegisteredTo.MathCourse$$

to specify that Math students are exactly those who are registered to Math courses only. However, if we define the concept *MathStudent* instead as follows:

$$MathStudent \equiv Student \sqcap \exists\ RegisteredTo.MathCourse$$

any student who is registered to at least one Math course will be considered as a Math student.

The inclusion axioms express relations between concepts. The simplest relations are the inclusion relations between *atomic* concepts or roles, which correspond to the `subClassOf` and to the `subPropertyOf` relations of RDFS. For example, if *MathCourse* and *Course* are atomic concepts in the vocabulary, and *LateRegisteredTo* and *RegisteredTo* are atomic roles in the vocabulary, the following inclusion axioms express that *MathCourse* is a more specific concept than (i.e., a subclass of) *Course*, and that *LateRegisteredTo* is a more specific role than (i.e., a subproperty of) *RegisteredTo*:

$$MathCourse \sqsubseteq Course$$
$$LateRegisteredTo \sqsubseteq RegisteredTo$$

General inclusion axioms (calleds GCIs) consist of inclusions between *complex* concepts. For example, the following GCI expresses the constraint that *only professors or lecturers may teach to undergraduate students* (which is expressible in OWL by a set of 5 triplets, as seen in Section 3.4):

$$\exists\ TeachesTo.Undergrad \sqsubseteq Professor \sqcup Lecturer$$

Such a constraint can interact with other constraints expressed in the Tbox, or in the Abox. For instance, suppose that we have in the Tbox (i.e., in the ontology) the following inclusion axioms stating that researchers are neither professors nor lecturers, that only students are taught to, and that students that are not undergraduate students are graduate students:

$Researcher \sqsubseteq \neg\ Professor$
$Researcher \sqsubseteq \neg\ Lecturer$
$\exists\ TeachesTo^{-} \sqsubseteq Student$
$Student \sqcap \neg\ Undergrad \sqsubseteq GraduateStudent$

Based on the logical semantics which will be detailed below, the following constraint can be inferred:

$Researcher \sqsubseteq \forall\ TeachesTo.GraduateStudent$

Suppose now that the Abox contains the following assertions on the individuals *dupond* and *pierre*:

TeachesTo(dupond,pierre)
$\neg$ *GraduateStudent(pierre)*

¬ *Professor(dupond)*
The new fact *Lecturer(dupond)* can be logically inferred from those facts and the above constraints in the Tbox.

The underlying reasoning leading to such inferences is quite elaborate and requires a complex algorithmic machinery to make it automatic. It is the focus of the remaining of this section.

FOL semantics of DL

Reasoning in DLs is based on the standard logical semantics in terms of FOL interpretations of individuals as constants, of concepts as subsets, and of roles as binary relations.

An *interpretation* consists of a nonempty *interpretation domain* Δ^I and an *interpretation function* I that assigns an element to each individual in **O**, a subset of Δ^I to each atomic concept **C** and a binary relation over Δ^I to each atomic role in **R**. Usually, in DL, the so called *unique name assumption* holds and thus *distinct* individuals are interpreted by *distinct* elements in the domain of interpretation.

The semantics of complex concepts using those constructs is recursively defined from the interpretations of atomic concepts and roles as follows:

- $I(C_1 \sqcap C_2) = I(C_1) \cap I(C_2)$
- $I(\forall R.C) = \{o_1 \mid \forall\ o_2\ [(o_1,o_2) \in I(R) \Rightarrow o_2 \in I(C)]\}$
- $I((\exists R.C) = \{o_1 \mid \exists o_2.[(o_1,o_2) \in I(R)\ \wedge\ o_2 \in I(C)]\}$
- $I(\neg C) = \Delta^I \setminus I(C)$
- $I(R^-) = \{(o_2,o_1) \mid (o_1,o_2) \in I(R)\}$

Satisfaction is defined as follows:

- An interpretation I *satisfies* (i.e., is a *model* of) an class inclusion axiom $B \sqsubseteq C$, resp. $B \equiv C$, if $I(B) \subseteq I(C)$, resp. $I(B) = I(C)$.
- I *satisfies* a relationship inclusion axiom $R \sqsubseteq E$, resp. $R \equiv E$, if $I(R) \subseteq I(E)$, resp. $I(R) = I(E)$.
- I *satisfies* the membership assertion $C(a)$, resp. $R(a,b)$, if $I(a) \in I(C)$, resp., $(I(a),I(b)) \in I(R)$.
- I *satisfies* or is *model of a knowledge base* $\mathcal{K} = \langle \mathcal{T},\mathcal{A} \rangle$ if it is a model of all the statements both in $\mathcal{T}$ and $\mathcal{A}$. A knowledge base $\mathcal{K}$ is *satisfiable* (or *consistent*) if it has at least one model.

Finally, a knowledge base $\mathcal{K}$ *logically entails* a (terminological or assertional) statement α, written $KB \models \alpha$, if every model of $\mathcal{K}$ is a also model of α.

Reasoning problems considered in DLs

The main reasoning problems that have been extensively studied in the DL community are satisfiability (i.e., consistency) checking of DL knowledge bases, and also instance checking and subsumption checking. They are formally defined as follows.

- *Satisfiability checking:* Given a DL knowledge base $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, is $\mathcal{K}$ satisfiable?
- *Subsumption checking:* Given a Tbox $\mathcal{T}$ and two concept expressions C and D, does $\mathcal{T} \models C \sqsubseteq D$?
- *Instance checking:* Given a DL knowledge base $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, an individual e and a concept expression C, does $\mathcal{K} \models C(e)$?

Instance checking and subsumption checking are logical entailment problems that can in fact be reduced to (un)satisfiability checking *for DLs having full negation in their language*, i.e., for DLs in which the constructor $\neg$ can apply to any complex concept in the axioms of the Tbox. The reason is that, based on the logical semantics, we have the following equivalences (in which a is a new individual that we introduce):

- $\mathcal{T} \models C \sqsubseteq D \Leftrightarrow \langle \mathcal{T}, \{(C \sqcap \neg D)(a)\} \rangle$ is unsatisfiable.
- $\langle \mathcal{T}, \mathcal{A} \rangle \models C(e) \Leftrightarrow \langle \mathcal{T}, \mathcal{A} \cup \{\neg C(e)\} \rangle$ is unsatisfiable.

For simple DLs in which the constructor of negation is not allowed, instance checking can be reduced to subsumption checking by computing the *most specific concept* satisfied by an individual in the Abox. Given an Abox $\mathcal{A}$ of a given DL and an individual e, the *most specific concept* of e in $\mathcal{A}$ (denoted $msc(\mathcal{A}, e)$) is the concept expression D in the given DL such that for every concept C in the given DL, $\mathcal{A} \models C(e)$ implies $D \sqsubseteq C$. Clearly, once $msc(\mathcal{A}, e)$ is known, we have:

$$\langle \mathcal{T}, \mathcal{A} \rangle \models C(e) \Leftrightarrow \mathcal{T} \models msc(\mathcal{A}, e) \sqsubseteq C$$

We now focus on some representative DLs. We start with $\mathcal{ALC}$ (Section 4.2) which is the basis of the most expressive DLs, and in particular those that led to OWL. Reasoning in these expressive DLs (and thus in OWL) is decidable but at the price of some high complexity often prohibitive in practice. We then survey DLs for which reasoning is polynomial: first $\mathcal{FL}$ and $\mathcal{EL}$ in Section 4.3, and finally the most recent DL-LITE family in Section 4.4, which provides a good trade-off between expressiveness and efficiency.

4.2 $\mathcal{ALC}$: the prototypical DL

$\mathcal{ALC}$ is often considered as the prototypical DL because it corresponds to a fragment of FOL that is easy to understand, and also because it is a syntactic variant of the basic modal logic K (see references). $\mathcal{ALC}$ is the DL based on the following constructs:

- *conjunction* $C_1 \sqcap C_2$,
- *existential restriction* $\exists R.C$,
- *negation* $\neg C$.

As a result, $\mathcal{ALC}$ also contains de facto:

- the *disjunction* $C_1 \sqcup C_2$ (which stands for $\neg(\neg C_1 \sqcap \neg C_2)$),
- the value restriction (since $\forall R.C$ stands for $\neg(\exists R.\neg C)$),
- the top $\top$ and bottom $\bot$ (standing respectively for $A \sqcup \neg A$ and $A \sqcap \neg A$).

An $\mathcal{ALC}$ Tbox may contain GCIs such as:

$$\exists TeachesTo.Undergrad \sqsubseteq Professor \sqcup Lecturer$$

An $\mathcal{ALC}$ Abox is made of a set of facts of the form $C(a)$ and $R(a,b)$ where a and b are individuals, R is an atomic role and C is a possibly complex concept.

Since $\mathcal{ALC}$ allows full negation, subsumption and instance checking in $\mathcal{ALC}$ can be trivially reduced to satisfiability checking of $\mathcal{ALC}$ knowledge bases, as seen previously.

The algorithmic method for reasoning in $\mathcal{ALC}$ (and in all expressive DLs extending $\mathcal{ALC}$) is based on tableau calculus, which is a classical method in logic for satisfiability checking. The *tableau method* has been extensively used in DLs both for proving decidability results and for implementing DL reasoners such as Fact, Racer and Pellet, respectively implemented in C++, Lisp-like, and in Java).

We just illustrate here the tableau method on a simple example, and refer the reader to the last section for pointers to more detailed presentations. Consider an $\mathcal{ALC}$ knowledge base whose Tbox $\mathcal{T}$ is without GCIs, i.e., $\mathcal{T}$ is made of concept definitions only. For instance:

$$\mathcal{T} = \{C_1 \equiv A \sqcap B, C_2 \equiv \exists R.A, C_3 \equiv \forall R.B, C_4 \equiv \forall R.\neg C_1\}$$

Let us consider the following associated Abox $\mathcal{A}$:

$$\mathcal{A} = \{C_2(a), C_3(a), C_4(a)\}$$

For checking whether the knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable, we first get rid of the Tbox by recursively unfolding the concept definitions. This is always possible for Tbox composed of a set of acyclic equivalence axioms of the form $A \equiv C$, where A is an atomic concept appearing in the left-hand side of exactly one equivalence axioms (no multiple definition). We obtain the following Abox which is equivalent to $\langle \mathcal{T}, \mathcal{A} \rangle$:

$$\mathcal{A}' = \{(\exists R.A)(a), (\forall R.B)(a), (\forall R.\neg(A \sqcap B))(a)\}$$

We now apply a preprocessing that consists in transforming all the concepts expressions in $\mathcal{A}'$ into *negation normal form* so that the negation construct applies to only atomic concepts. This transformation can be done in polynomial time. The result is the equivalent following Abox $\mathcal{A}''$:

$$\mathcal{A}'' = \{(\exists R.A)(a), (\forall R.B)(a), (\forall R.(\neg A \sqcup \neg B))(a)\}$$

The tableau method tries to build a finite model of $\mathcal{A}''$ by applying tableau rules to extend it. There is one rule per construct (except for the negation construct). From an extended Abox which is *complete* (no rule applies) and *clash-free* (no obvious contradiction), a so-called *canonical interpretation* can be built, which is a model of the initial Abox.

More precisely, the tableau rules applies to a set of Aboxes, starting from $\{\mathcal{A}''\}$. The rules picks one Abox and replaces it by finitely many new Aboxes. New Aboxes containing a clash (i.e., two contradictory facts $A(e)$ and $\neg A(e)$) are simply deleted. The algorithm terminates if no more rules apply to any Abox in the set. The returned answer is then *yes* (the input Abox is satisfiable) if the set is not empty, and *no* otherwise.

The tableau rules for $\mathcal{ALC}$ (applied to an Abox $\mathcal{A}$ in the set of Aboxes) are the following:

- *The* $\sqcap$ *-rule*:
  Condition: $\mathcal{A}$ contains $(C \sqcap D)(a)$ but not both $C(a)$ and $D(a)$
  Action: add $\mathcal{A}' = \mathcal{A} \cup \{C(a), D(a)\}$
- *The* $\sqcup$ *-rule*:
  Condition: $\mathcal{A}$ contains $(C \sqcup D)(a)$ but neither $C(a)$ nor $D(a)$
  Action: add $\mathcal{A}' = \mathcal{A} \cup \{C(a)\}$ and $\mathcal{A}'' = \mathcal{A} \cup \{D(a)\}$
- *The* $\exists$ *-rule*:
  Condition: $\mathcal{A}$ contains $(\exists R.C)(a)$ but there is no c such that $\{R(a,c), C(c)\} \subseteq \mathcal{A}$
  Action: add $\mathcal{A}' = \mathcal{A} \cup \{R(a,b), C(b)\}$ where b is a new individual name
- *The* $\forall$ *-rule*:
  Condition: $\mathcal{A}$ contains $(\forall R.C)(a)$ and $R(a,b)$ but not $C(b)$
  Action: add $\mathcal{A}' = \mathcal{A} \cup \{C(b)\}$

The result of the application of the tableau method to $\mathcal{A}'' = \{(\exists R.A)(a), (\forall R.B)(a), (\forall R.(\neg A \sqcup \neg B))(a)\}$ gives the following Aboxes:

$$\mathcal{A}''_1 = \{(\exists R.A)(a), (\forall R.B)(a), (\forall R.(\neg A \sqcup \neg B))(a), R(a,b), A(b), B(b), \neg A(b)\}$$

$$\mathcal{A}''_2 = \{(\exists R.A)(a), (\forall R.B)(a), (\forall R.(\neg A \sqcup \neg B))(a), R(a,b), A(b), B(b), \neg B(b)\}$$

They both contain a clash. Therefore, the original $\mathcal{A}''$ is correctly decided unsatisfiable by the algorithm.

The interest of the tableau method is that it is "easily" extensible to new constructs and new constraints. For instance, in order to extend the previous tableau method to $\mathcal{ALC}$ with GCIs, we first observe that a finite set of GCIs $\{C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n\}$ can be encoded into one GCI of the form $\top \sqsubseteq C$ where C is obtained by transforming $(\neg C_1 \sqcup D_1) \sqcap \ldots \sqcap (\neg C_n \sqcup D_n)$ in negation normal form, and we add the following tableau rule:

The GCI *-rule* for $\top \sqsubseteq C$:
Condition: $\mathcal{A}$ contains the individual name a but not $C(a)$
Action: add $\mathcal{A}' = \mathcal{A} \cup \{C(a)\}$

The subtle point is that by adding this rule, the termination of the tableau method is not guaranteed, as it can be seen just by considering the Abox $\{P(a)\}$ and the GCI $\top \sqsubseteq \exists R.P$. The clue is to add a blocking condition for stopping the generation of new individual names and to prevent the tableau rules for applying to blocked individuals. An individual y is blocked by an individual x such as the set of concepts describing y is included in the set of concepts describing x. In our example, from the Abox obtained by applying the GCI rule to $\{P(a)\}$, we stop at the clash-free Abox $\mathcal{A} = \{P(a), (\exists R.P)(a), R(a,b), P(b), (\exists R.P)(b), R(b,c), P(c)\}$ since the individual c is blocked by the individual b. The canonical interpretation I of a clash-free Abox $\mathcal{A}_n$ to which no more rules applies is obtained by defining as domain of interpretation Δ^I the set of all the individual appearing in the corresponding Abox and

- for each atomic concept A: $I(A) = \{e \in \Delta^I \mid A(e) \in \mathcal{A}_n\}$
- for each atomic role R: $I(R) = \{(e,f) \in \Delta^I \times \Delta^I \mid R(e,f) \in \mathcal{A}_n\} \cup \{R(f,f) \mid f$ is blocked by e such that $R(e,f) \in \mathcal{A}_n\}$

It can be shown that this canonical interpretation is in fact a model of the corresponding clash-free Abox, and therefore of the original Abox which is therefore satisfiable.

The tableau method shows that the satisfiability of $\mathcal{ALC}$ knowledge bases is decidable but with a complexity that may be exponential because of the disjunction construct and the associated $\sqcup$-*rule*.

4.3 Simple DLs for which reasoning is polynomial

$\mathcal{FL}$ and $\mathcal{EL}$ are two minimal DLs for which subsumption checking is polynomial for Tboxes without GCIs. For such simple Tboxes, as already mentioned previously, by concept unfolding, we can get rid of the Tbox and the subsumption checking problem becomes: given two concept expressions C and D, does $\models C \sqsubseteq D$?, i.e., for any individual x, does $C(x)$ implies $D(x)$?

The constructs allowed in $\mathcal{FL}$ are *conjunction* $C_1 \sqcap C_2$, *value restrictions* $\forall R.C$ and also *unqualified existential restriction* $\exists R$. Satisfiability is trivial in $\mathcal{FL}$: every $\mathcal{FL}$ knowledge base is satisfiable. Subsumption checking between two concept expressions C and D can be done in quadratic time by a *structural subsumption* algorithm $IsSubsumed?(C,D)$, which consists in:

- Normalizing the concept expressions. The normal form of a $\mathcal{FL}$ concept expression is obtained by:
  - flattening all nested conjunctions, i.e., by applying exhaustively the rewriting rule to the concept expression: $A \sqcap (B \sqcap C) \rightarrow A \sqcap B \sqcap C$
  - pushing value restrictions over conjunctions, i.e., by applying exhaustively the rewriting rule to the concept expression: $\forall R.(A \sqcap B) \rightarrow \forall R.A \sqcap \forall R.B$.

  For example, the normalization of $\forall R.(A \sqcap \forall S.(B \sqcap \forall R.A))$ returns the expression: $\forall R.A \sqcap \forall R.\forall S.B \sqcap \forall R.\forall S.\forall R.A$.

  The application of these rewriting rules preserves logical equivalence, hence the subsumption is preserved by the normalization. (Proof left in exercise).

- Comparing recursively the structure of the normalized expressions $C_1 \sqcap \ldots \sqcap C_n$ and $D_1 \sqcap \ldots \sqcap D_m$ as follows: $IsSubsumed?(C,D)$ return *true* if and only if for all D_i
  - if D_i is an atomic concept or an unqualified existential restriction, then there exists a C_j such that $C_j = D_i$
  - if D_i is a concept of the form $\forall R.D'$, then there exists a C_j of the form $\forall R.C'$ (same role) such that $IsSubsumed?(C',D')$

By induction on the number of nesting of the $\forall$ constructs, it is easy to prove that the above algorithm runs in $O(|C| \times |D|)$ where $|C|$ denotes the size of the concept expression C defined by the number of the constructs $\sqcap$ and $\forall$ appearing in it.

The structural subsumption is *sound* since when $IsSubsumed?(C,D)$ returns true, then it holds that $I(C) \subseteq I(D)$ for every interpretation I. Take any conjunct D_i of D; either it appears as a conjunct C_j of C and by definition of the logical semantics of the conjunction construct: $I(C) \subseteq I(C_j) = I(D_i) \subseteq I(D)$; or it is of the form $\forall R.D'$ and there exists as conjunct of C of the form $\forall R.C'$ such that $IsSubsumed?(C', D')$; then, by induction $I(C') \subseteq I(D')$, and by definition of the logical semantics of the $\forall$ construct: $I(\forall R.C') \subseteq I(\forall R.D')$, and thus by the conjunction semantics, $I(C) \subseteq I(C_j) \subseteq I(D_i) \subseteq I(D)$.

The *completeness* of the structural subsumption is a little bit harder to prove: it must be shown that, whenever $I(C) \subseteq I(D)$ for all interpretations I, then the algorithm $IsSubsumed?(C, D)$ returns $true$. The proof is done by contrapositive, i.e., by showing that anytime $IsSubsumed?(C, D)$ returns $false$, then there exists an interpretation assigning an element of the domain to C and not to D, i.e., $C \not\sqsubseteq D$. The proof relies on the fact that anytime $IsSubsumed?(C, D)$ returns $false$, there exists a conjunct D_i of D which has no correspondent conjunct in C. In this case, we can build an interpretation I in which all the conjuncts in C are assigned to subsets containing a given element e, and in which D_i is assigned to the empty set: $e \in I(C)$ and $e \notin I(D)$.

As an exercise, by applying the above algorithm, check that:

$$\forall R.(\forall S.B \sqcap \forall S.\forall R.A) \sqcap \forall R.(A \sqcap B) \sqsubseteq \forall R.(A \sqcap \forall S.(B \sqcap \forall R.A))$$

For $\mathcal{FL}$ general Tboxes, i.e., Tboxes including general concept inclusions (GCIs), subsumption checking becomes intractable even for Tboxes containing inclusion axioms between atomic concepts only. In this case, subsumption checking is co-NP complete (by reduction from the inclusion problem for acyclic finite automata).

The constructs allowed in $\mathcal{EL}$ are *conjunctions* $C_1 \sqcap C_2$ and *existential restrictions* $\exists R.C$. Like for $\mathcal{FL}$, any $\mathcal{EL}$ knowledge base is satisfiable. Subsumption checking in $\mathcal{EL}$ is polynomial even for general Tboxes, i.e., Tboxes including general concept inclusions (GCIs). The subsumption algorithm for $\mathcal{EL}$ can also be qualified as a structural algorithm, although it is quite different from the "normalize and compare" algorithm for $\mathcal{FL}$ concept expressions described previously. In fact, it relies on the representation of $\mathcal{EL}$ concept expressions as labeled trees (called description trees), in which nodes are labeled with sets of atomic concepts, while edges are labeled with atomic roles. It is shown that an $\mathcal{EL}$ concept expression C is subsumed by an $\mathcal{EL}$ concept expression D if there is an homomorphism from the description tree of D to the description tree of C. Checking subsumption corresponds then to checking the existence of an homomorphism between trees. This problem is known to be NP-complete for graphs but to be polynomial for trees. Taking into account GCIs in the Tbox can be done by extending accordingly the labels of the description trees.

Therefore, if we can say for short that **subsumption checking is polynomial for** $\mathcal{EL}$, we have to be more cautious for $\mathcal{FL}$: we just can say that subsumption checking between two $\mathcal{FL}$ concept expressions (w.r.t. an empty Tbox) is polynomial.

As explained previously, instance checking can be reduced to subsumption checking by computing the *most specific concept* of a constant e in $\mathcal{A}$ (denoted $msc(\mathcal{A}, e)$). The problem is that in $\mathcal{FL}$ or $\mathcal{EL}$ the most specific concepts do not always exist. A solution for checking whether $C(e)$ is entailed from an $\mathcal{FL}$ or $\mathcal{EL}$ knowledge base is to adapt the tableau method: first, the tableau rules corresponding to the constructs allowed in $\mathcal{FL}$ and $\mathcal{EL}$ can be applied to saturate the original Abox; then, the negation of $C(e)$ is injected and the tableau rules are applied, including possibly the $\sqcup$-rule, since the $\sqcup$ construct can be introduced by negating $\mathcal{FL}$ or $\mathcal{EL}$ concept expressions; $C(e)$ is entailed from the original knowledge base if and only if all the resulting Aboxes contain a clash.

If we combine the constructs of $\mathcal{FL}$ and $\mathcal{EL}$, namely *conjunction* $C_1 \sqcap C_2$, *value restrictions* $\forall R.C$, and *existential restrictions* $\exists R.C$, we obtain the new DL called $\mathcal{FLE}$ for which even checking subsumption between two concept expressions is NP-complete.

In fact, since the DL *existential restrictions* and *value restrictions* correspond to the OWL restrictions **owl:oneValuesFrom** and **owl:allValuesFrom** (see Figure 8), that means that the combination of those restrictions that are quite natural from a modeling point of view may lead to intractability for automatic inferencing.

4.4 The DL-LITE family: a good trade-off

The DL-LITE family has been recently designed for capturing the main modeling primitives of conceptual data models (e.g., Entity-Relationship) and object-oriented formalisms (e.g., basic UML class diagrams[1]), while remaining reasoning tractable in presence of concept inclusion statements and a certain form of negation.

The constructs allowed in DL-LITE are *unqualified existential restriction* on roles ($\exists R$) and on inverse of roles ($\exists R^-$), and the negation.

The axioms allowed in a Tbox of DL-LITE are concept inclusion statements of the form $B \sqsubseteq C$ or $B \sqsubseteq \neg C$, where B and C are atomic concepts, or existential restriction ($\exists R$ or $\exists R^-$).

DL-LITE$_\mathcal{F}$ and DL-LITE$_\mathcal{R}$ are then two dialects of DL-LITE that differ from some additional allowed axioms:

- a DL-LITE$_\mathcal{R}$ Tbox allows role inclusion statements of the form $P \sqsubseteq Q$ or $P \sqsubseteq \neg Q$, where P and Q are atomic roles or inverse of atomic roles
- a DL-LITE$_\mathcal{F}$ Tbox allows functionality statements on roles of the form $(funct\ P)$ or $(funct\ P^-)$. An interpretation I *satisfies* a functionality statement $(funct\ R)$ if the binary relation $I(R)$ is a function, i.e., $(o, o_1) \in I(R)$ and $(o, o_2) \in I(R)$ implies $o_1 = o_2$.

It is worth noticing that negation is only allowed in right hand sides of inclusion statements. Inclusion axioms with negation in the right-hand side are called *negative inclusions* (for short *NIs*), while the inclusion axioms without negation are called *positive inclusions* (for short *PIs*).

It can be shown that subsumption checking is polynomial both for DL-LITE$_\mathcal{R}$ and DL-LITE$_\mathcal{F}$ Tboxes, and that their combination (denoted DL-LITE$_{\mathcal{RF}}$) is PTIME-complete. The subsumption algorithm is based on computing the *closure* of the Tbox, i.e., the set of all PIs and NIs that can be inferred from the original Tbox. Checking $\mathcal{T} \models C \sqsubseteq D$ consists then in checking whether $C \sqsubseteq D$ belongs to the closure of $\mathcal{T}$. Satisfiability checking and instance checking also rely on exploiting the closure. In fact, they are particular cases of the most general problem of answering conjunctive queries over DL-LITE knowledge bases, for which the DL-LITE family has been designed. Answering conjunctive queries over ontologies is a reasoning problem of major interest for the Semantic Web, the associated decision problem of which *is not reducible* to (un)satisfiability checking or to subsumption or instance checking. This problem has been studied quite recently and we will dedicate a whole chapter to it (Chapter **??**). In particular, we will see that the DL-lite family groups DLs that have been specially designed for guaranteeing query answering to be polynomial in data complexity.

It is noticeable that DL-LITE$_\mathcal{R}$ has been recently incorporated into the version OWL2 of OWL as the profile called OWL2 QL. This profile is an extension of RDFS.

[1]See http://www.omg.org/uml

5 Further reading

We refer to existing books (e.g., [AH08, AvH08]) for a full presentation of RDF, RDFS and OWL.

We use an abstract compact syntax for RDF, RDFS and OWL statements, instead of their verbose XML notation. The usage of the XML syntax for RDF, RDFS or OWL is mainly for exchanging metadata and ontologies in a standard format. XML name spaces and XML tools for parsing, indexing, transforming, browsing can be used for that purpose. Several tools such as Jena [Jen] are now widely available and used to store, manage and query RDFS data. We will discuss a query language for RDF, namely SPARQL, in Chapter **??**.

RDFS (without blank nodes) can be seen as a fragment of $DL\text{-}Lite_R$, which is a DL of the $DL\text{-}Lite$ family, described in [CGL+07]. The $DL\text{-}Lite$ family has been designed for enabling tractable query answering over data described w.r.t. ontologies. We will consider the issue of querying with ontologies in Chapter **??**.

The readers interested by the translation in FOL of the full RDFS (possibly with blank nodes in place of properties) are referred to [BFT05].

A set of RDFS triplets can also be formalized in conceptual graphs [CM08] that are graphical knowledge representation formalisms that have a FOL semantics for which reasoning can be performed by graph algorithms (such as projection). The formalization of RDFS in conceptual graphs have been studied in [BCG+10].

At the moment, in contrast with RDFS, there is very little usage of tools supporting reasoning over OWL statements. The only available reasoners are description logic reasoners such as Fact [FAC], RACER [RAC] or Pellet [SPG+07].

Readers interested by a comprehensive summary of the complexity results and reasoning algorithms on Description Logic are referred to [BCM+03]. There is a close relation between some Description Logic (in particular $\mathcal{ALC}$) and modal logics [BBW06] which have been extensively studied in Artificial Intelligence.

Satisfiability checking in $\mathcal{ALC}$ (and thus also subsumption and instance checking) has been shown EXPTIME-complete. Additional constructs like those in the fragment OWL DL of OWL which corresponds to DLs [2] do not change the complexity class of reasoning (which remains EXPTIME-complete). In fact, OWL DL is a syntactic variant of the so-called $\mathcal{SHOIN}$ DL, which is obtained from $\mathcal{ALC}$ by adding *number restrictions* $(\geq nP)$, *nominals* $\{a\}$, and *inverse roles* P^- of atomic roles. Nominals make it possible to construct a concept representing a singleton set {a} (a nominal concept) from an individual a. In addition, some atomic roles can be declared to be *transitive* using a role axiom $Trans(P)$, and the Tbox can include *role inclusion* axioms $R_1 \sqsubseteq R_2$.

The semantics of those additional constructs and axioms is defined from the interpretations of individuals, atomic concepts and roles as follows ($\sharp\{S\}$ denotes the cardinality of a set S):

- $I((\geq nP)) = \{d \in \Delta^I \mid \sharp\{e \mid (d,e) \in I(P)\} \geq n\}$
- $I(P^-) = \{(o_2,o_1) \mid (o_1,o_2) \in I(P)\}$
- $I(\{a\}) = \{I(a)\}$

An interpretation I *satisfies* a role transitivity statement $(Trans\, P)$ if the binary relation $I(P)$ is transitive, i.e., $(o,o_1) \in I(P)$ and $(o_1,o_2) \in I(P)$ implies $o = o_2$.

[2]OWL Full is undecidable

$\mathcal{SHIQ}$ extends $\mathcal{SHOIN}$ with so-called *qualified number restrictions* $(\geq nP.C)$ whose semantics is defined by the following interpretation rule:

$$I((\geq n\,P.C)) = \{d \in \Delta^I \mid \sharp\{e \mid (d,e) \in I(P) \wedge e \in I(C)\} \geq n\}.$$

Relating the ontology modeling OWL language to DLs is of primary importance since it allows to understand the cost to pay if we want to automatically exploit constraints requiring a given combination of OWL constructs. In particular, we know (from the EXPTIME-completeness of $\mathcal{ALC}$) that *in the worst case* an inference algorithm may take an exponential time for reasoning on a set of constraints expressed using full negation, conjunction, existential and value restriction. In practice however, the existing DL reasoners such as FaCT, RACER and Pellet have acceptable performances for dealing with expressive ontologies of reasonable size. The reason is that the constraints expressed in real-life ontologies usually do not correspond to the pathological cases leading to the worst-case complexity.

6 Exercises

Exercise 6.1 *Show that the inference rules of Table 7 are complete for the logical entailment of constraints expressible in RDFS. More precisely, let $\mathcal{T}$ be a set of RDFS triplets expressing logical constraints of the form $A \sqsubseteq B$, $P \sqsubseteq R$, $\exists P \sqsubseteq A$ or $\exists P^- \sqsubseteq B$, where A and B denote classes while P and Q denote properties. Show that for every constraint $C \sqsubseteq D$ of one of those four previous forms, if $\mathcal{T} \models C \sqsubseteq D$ then the RDFS triplet denoting $C \sqsubseteq D$ is inferred by applying the inference rules of Table 7 to $\mathcal{T}$.*

Exercise 6.2 *$\mathcal{AL}$ is the Description Logic obtained from $\mathcal{FL}$ by adding negation on atomic concepts.*

1. *Based on the logical semantics, prove that the concept expression $\forall R.A \sqcap \forall R.\neg A$ is subsumed by the concept expression $\forall R.B$, for any atomic concept B.*
2. *Show that the structural subsumption algorithm is not complete for checking subsumption in $\mathcal{AL}$.*

   Indication: *apply the algorithm IsSubsumed?$(\forall R.A \sqcap \forall R.\neg A, \forall R.B)$.*
3. *Apply the tableau method for checking that $\forall R.A \sqcap \forall R.\neg A$ is subsumed by the concept expression $\forall R.B$.*

   Indication: *apply the tableau rules to the Abox $\{(\forall R.A \sqcap \forall R.\neg A)(a), (\exists R.\neg B)(a)\}$*

Exercise 6.3 *Based on the logical semantics, show the following statements:*

1. $\forall R.(A \sqcap B) \equiv \forall R.A \sqcap \forall R.B.$
2. $\exists R.(A \sqcap B) \not\equiv \exists R.A \sqcap \exists R.B$
3. $\exists R.(A \sqcap B) \sqsubseteq \exists R.A \sqcap \exists R.B$
4. $\exists R.A \sqcap \forall R.B \sqsubseteq \exists R.(A \sqcap B)$

[AH08] D. Allemang and J. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL.* Morgan-Kaufman, 2008.

[AvH08] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. The MIT Press, 2008.

[BBW06] P. Blackburn, J. Van Benthem, and F. Wolter. *Handbook of Modal Logic*. Springer, 2006.

[BCG$^+$10] J.-F. Baget, M. Croitoru, A. Gutierrez, M. LeclÃ¨re, and M.-L. Mugnier. Translations between rdf(s) and conceptual graphs. In *Proc. Intl. Conference on Conceptual Structures (ICCS)*, pages 28–41, 2010.

[BCM$^+$03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BFT05] Jos De Bruijn, Enrico Franconi, and Sergio Tessaris. Logical reconstruction of normative RDF. In *Proc. OWL: Experiences and Directions Workshop (OWLED'05)*, 2005.

[CGL$^+$07] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-LITE Family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[CM08] Michel Chein and Marie-Laure Mugnier. *Graph-based Knowledge Representation*. Springer, 2008.

[FAC] FaCT++. `http://owl.cs.manchester.ac.uk/fact++/`.

[Jen] Jena - a semantic web framework for java. http://jena.sourceforge.net/.

[RAC] Racerpro. http://www.racer-systems.com/.

[SPG$^+$07] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.