Knowledge representation and semantic technologies - OWL

Claudia Medaglia, Sveva Pepe

Sunday 20th December, 2020

1 Introduction

1.1 Evaluation

The semantic of a SPARQL query of a SELECT statement consists of the set of answers that are produced by the satisfaction of the WHERE conditions that is made by finding a match of the condition's graph pattern with the real graph. → **Simple Entailment Regime**
OK for SPARQL 1.0 but not with SPARQL 1.1.
There are special symbols in a RDF graph that have a special semantics/meaning that are defined in the RDF/RDFS standars itself (es. rdf:subject, rdf:predicate, etc.).
These special symbols are considered special by SPARQL 1.1, while for SPARQL 1.0 not, it sees everything in the RDF graph like a simple graph (missing the special meaning of these special symbols of the RDF/RDFS vocabulary).

1.2 RDF, RDFS, OWL and SPARQL

We are going to add a new RDF vocabulary for ontology (after RDF and RDFS): the OWL vocabulary.
At the end, SPARQL will be the query language for ontology (that is a DL knowledge base).
OWL is a concrete syntax for DL knowledge bases and SPARQL has a special semantics devoted to the interaction with RDF graph representing DL knowledge bases.

RDF: is a language for writing graphs that uses URIs as labels of nodes and edges, and talks about Web identifier. Then the RDF language has a set of predefined and special meaning URIs(es rdf:subject, rdf:type, etc.) that should be addressed by SPARQL language, while in the simple semantics, that is only based on graph matches, this aspect that are special names in the RDF vocabulary is not address. → First semantic problem.

RDF has a correlate standard, the RDFS, that is essentially just another vocabulary so it enriches the special names with RDFS URIs that have a special role (es. used to define classes, subclasses, range, etc.)
Also the special meaning of this RDFS vocabulary is not addressed by simple SPARQL evaluation → Second semantic problem.

Third semantic problem → Also OWL is another vocabulary for RDF, just like RDFS.
At the end the ontology layer is written in RDF using this new vocabulary.
So we have 3 special vocabularies in a RDF graph that can be used and the query language for this type of things is SPARQL.
But SPARQL should be provided with a means to correctly address the semantics of the special symbols of these 3 vocabularies: SPARQL 1.1 has tried to really take into account the semantics problems introduced by the URIs with special semantics coming from RDF, RDFS and OWL vocabulary.
(We are not going to leave the RDF language because in the end an OWL resource/ontology will be represented by a RDF graph and this is why SPARQL is a good query language of OWL..... we will see it later).

2 Ontologies and OWL

The OWL layer is the ontology layer and is conceptually considered on top of the RDF layer. (Here DL is coming to play).
Its purpose is to really provide semantics to the specifications.
RDF provides concrete syntax for annotation of Web Resources.
RDFS provides a way to define a simple TBox, that allows to write semantics relationships between terms/ids/names/symbols (when you say that x is a subclass of y for example).

Ontology vocabulary introduces a more expressive language w.r.t RDFS to write semantics relationships between URIs in the graph.

2.1 Ontology in computer science

- ontology = shared conceptualization of a domain of interest
  - conceptual model means abstract model
  - shared means that the usage of the terms in this conceptualization should be general enough to be naturally accepted from a vaste domain of users

  Different types of ontology:

  - Simple ontology: shared vocabulary of terms that should be understood and shared by a big domain of users
  - Complex (deep) ontology: beside the terms and their meaning, we can talk about a more complex ontology that defines relationships between terms
  - Taxonomy
  - AI view: logical theory (knowledge base)
  - DB view: conceptual data model

We want to look for formalization to the notion of shared conceptual model of a domain of interest.
But why we need a shared conceptual model? Because the goal of the semanitc Web is to annotate URIs with semantics in a form that is automatically understandable by tools.
So in order to express relation between the symbols that we are using, we have to use a knowledge representation language, in particular **DL** (because it is a class-based representation language, and RDF and RDFS already use classes and properties. RDF triples are role assertions, (subj, pred, obj), they are really binary associations; a graph is a binary relation, and a binary relation is a role. So we can express a RDF graph as a set of role assertions) In general, there are a lot of ontology langauges, like HTML and XML (we know their limits).
Even RDFS is an ontology language, but is too simple to describe resources in sufficient (no localised range and domain constraints, no existence/cardinality constraints, no transitive, inverse or symmetrical properties) detail and to express sofisticated semantics relationships (we have seen how expressive is DL w.r.t the simple statements that we can write with RDFS) 4→ Low expressive power.
At the same time, expressiveness has a price to pay when we process the information: RDFS has better performances (faster) than DL that is harder to process.

2.1.1 OWL versions

The OWL family is constituted by 3 different languages (with different expressive power):
- **OWL Full:** union of OWL syntax and RDF (DL + RDF). Here there is a theorem that proves that in some situation it is impossible to get an answer for the basic reasoning task in a finite amount of time. It does not admit terminting automatically reasoning techniques.
- **OWL-DL:** "DL-fragment" of OWL Full (here we renounce to some abilities and features of RDF: some special preficates, the fact that same symbols can play multiple roles in the same specification, etc. Here everything that we write in OWL can be translate in DL, it is not the same in OWL Full). It is the most used of the three, because it is more expressive (even if OWL-Lite has better performance).
- **OWL-Lite:** "easier to implement" subset of OWL (it allows automatic reasoning techniques).

2.2 OWL class constructors

Now we are going to talk just of OWL-DL. It is a concrete syntax for DL.
The DL that is considered in OWL is SHOIN(Dn), it is more expressive than ALC.
Here we have the same constructors we have in ALC, but in addition we also have:
- oneOf: it allows to write singleton concepts x_i constituted just by one individual and the oneOf builds the union of the individuals, so it is the concept constituted by n individuals whose instances are exactly the n individuals x_1,...,x_n.
-maxCardinality and min Cardinality: we can create the class of individuals that has at most n partecipation to a propery or at least n partecipation to a propery.

Also we have the XMLS **datatypes** (es. integer, boolean, string, etc.), so in this way we can talk about

values like integers, strings, a more detailed domain of interpretation, there are still abstract objects but also concrete objects (that we don't have in DL).
As in ALC we can have arbitrarily complex nesting of constructors.

Constructor	DL Syntax	Example	Modal Syntax
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	Human ⊓ Male	$C_1 \wedge \ldots \wedge C_n$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	Doctor ⊔ Lawyer	$C_1 \vee \ldots \vee C_n$
complementOf	$\neg C$	¬Male	$\neg C$
oneOf	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	{john} ⊔ {mary}	$x_1 \vee \ldots \vee x_n$
allValuesFrom	$\forall P.C$	∀hasChild.Doctor	$[P]C$
someValuesFrom	$\exists P.C$	∃hasChild.Lawyer	$\langle P \rangle C$
maxCardinality	$\leqslant nP$	⩽1hasChild	$[P]_{n+1}$
minCardinality	$\geqslant nP$	⩾2hasChild	$\langle P \rangle_n$

- XMLS datatypes as well as classes in ∀P.C and ∃P.C
  - E.g., ∃hasAge.nonNegativeInteger
- Arbitrarily complex nesting of constructors
  - E.g., Person ⊓ ∀hasChild.Doctor ⊔∃hasChild.Doctor

2.3 OWL axioms

Remember than in ALC we have just one type of axiom in the TBox, only concept inclusion statement or axiom. Here the concept inclusion is the subClassOf axiom.
Since this language is more expressive than ALC, in addition we have a lot axiom types.

Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human ⊑ Animal ⊓ Biped
equivalentClass	$C_1 \equiv C_2$	Man ≡ Human ⊓ Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male ⊑ ¬Female
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	{President_Bush} ≡ {G_W_Bush}
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	{john} ⊑ ¬{peter}
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter ⊑ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost ≡ price
inverseOf	$P_1 \equiv P_2^-$	hasChild ≡ hasParent$^-$
transitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+$ ⊑ ancestor
functionalProperty	$\top \sqsubseteq \leqslant 1P$	⊤ ⊑ ⩽1hasMother
inverseFunctionalProperty	$\top \sqsubseteq \leqslant 1P^-$	⊤ ⊑ ⩽1hasSSN$^-$

Axioms (mostly) reducible to inclusion (⊑)

C ≡ D iff both C ⊑ D and D ⊑ C

- transitiveProperty = the propery contains its own tranistive closure
- functionalProperty = " everybody has at least one mother"

Differences: In ACL we have just one type of role.
In OWL there are two kinds of roles (that became properties because we are in the Web Semantics context): object properties, properties that relate astract objects to abstract objects, and data properties, properties that relate generic abstract objects/individual to data values/concrete data type (es age). There are three different predicates: classes, object properties and data properties.

Remember that at the end everything is translated in XML.

E.g., concept Person ⊓ ∀hasChild.Doctor ⊔∃hasChild.Doctor:

```
<owl:Class>
  <owl:intersectionOf rdf:parseType=" collection">
    <owl:Class rdf:about="#Person"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild"/>
      <owl:toClass>
        <owl:unionOf rdf:parseType=" collection">
          <owl:Class rdf:about="#Doctor"/>
          <owl:Restriction>
            <owl:onProperty rdf:resource="#hasChild"/>
            <owl:hasClass rdf:resource="#Doctor"/>
          </owl:Restriction>
        </owl:unionOf>
      </owl:toClass>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

2.4 OWL functional and RDF syntax

The **functional syntax** is the easiest way to write statements in OWL. It uses a functional-style syntax, which means we write the constructors and the statements as functions having arguments between parentesis but not

separeted with commas (just by blank space).Es. DisjointClasses(C_1 C_2).
The **RDF syntax**: it is the RDF representation of OWL. We know that RDF is the level below the ontology layer, and at the end everything we write in the ontology layer is translated in RDF, and it is possible by introducing a new vocabulary for RDF, the OWL vocabulary. So we still write triples but these triples contains special predicates/identifiers that come from the OWL vocabulary. For instance the previous statement can be expressed in RDF in this way: C_1 owl:disjointWith C_2 (using the special OWL property "disjointWith" that has a special semantics and programs that process these graphs must be aware of this).

2.5 Abbreviations

Letters	**Meaning**	**Letters**	**Meaning**	**Letters**	**Meaning**	**Letters**	**Meaning**
C	class expression	CN	class name	D	data range	DN	datatype name
P	object property expression	PN	object property name	R	data property	A	annotation property
a	individual	aN	individual name	_:a	anonymous individual (a blank node label)	v	literal
n	non-negative integer	f	facet	ON	ontology name	U	IRI
s	IRI or anonymous individual	t	IRI, anonymous individual, or literal	p	prefix name	_:x	blank node
(a_1 … a_n)	RDF list						

2.6 Delarations

Language Feature	Functional Syntax	RDF Syntax
class	Declaration(Class(CN))	CN rdf:type owl:Class.
datatype	Declaration(Datatype(DN))	DN rdf:type rdfs:Datatype.
object property	Declaration(ObjectProperty(PN))	PN rdf:type owl:ObjectProperty.
data property	Declaration(DataProperty(R))	R rdf:type owl:DatatypeProperty.
annotation property	Declaration(AnnotationProperty(A))	A rdf:type owl:AnnotationProperty.
named individual	Declaration(NamedIndividual(aN))	aN rdf:type owl:NamedIndividual.

We essentially move from a functional notation to a triple notation.

2.7 Boolean operators and enumeration

Language Feature	Functional Syntax	RDF Syntax
intersection	ObjectIntersectionOf(C_1 … C_n)	_:x rdf:type owl:Class. _:x owl:intersectionOf (C_1 … C_n).
union	ObjectUnionOf(C_1 … C_n)	_:x rdf:type owl:Class. _:x owl:unionOf (C_1 … C_n).
complement	ObjectComplementOf(C)	_:x rdf:type owl:Class. _:x owl:complementOf C.
enumeration	ObjectOneOf(a_1 … a_n)	_:x rdf:type owl:Class. _:x owl:oneOf (a_1 … a_n).

2.8 Object property restrictions

Language Feature	Functional Syntax	RDF Syntax
universal	ObjectAllValuesFrom(P C)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:allValuesFrom C
existential	ObjectSomeValuesFrom(P C)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:someValuesFrom C
individual value	ObjectHasValue(P a)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:hasValue a.
local reflexivity	ObjectHasSelf(P)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:hasSelf "true"^^xsd:boolean.
exact cardinality	ObjectExactCardinality(n P)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:cardinality n.
qualified exact cardinality	ObjectExactCardinality(n P C)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:qualifiedCardinality n. _:x owl:onClass C.

- **individual value**: it is similar to the previous one but "a" is just an individual value/ name. Es ObjectHasValue(LivesIn Rome), I'm denoting all the class expression of all the individuals that paretecipate in the role LivesIn and the associated object is Rome. (It qualify the partecipation to the role of an individual, not of a concect like in the previous case).

Language Feature	Functional Syntax	RDF Syntax
maximum cardinality	ObjectMaxCardinality(n P)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:maxCardinality n.
qualified maximum cardinality	ObjectMaxCardinality(n P C)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:maxQualifiedCardinality n. _:x owl:onClass C.
minimum cardinality	ObjectMinCardinality(n P)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:minCardinality n.
qualified minimum cardinality	ObjectMinCardinality(n P C)	_:x rdf:type owl:Restriction. _:x owl:onProperty P. _:x owl:minQualifiedCardinality n. _:x owl:onClass C.

2.9 Data property restrictions

Language Feature	Functional Syntax	RDF Syntax
universal	DataAllValuesFrom(R D)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:allValuesFrom D.
existential	DataSomeValuesFrom(R D)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:someValuesFrom D.
literal value	DataHasValue(R v)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:hasValue v.
exact cardinality	DataExactCardinality(n R)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:cardinality n.
qualified exact cardinality	DataExactCardinality(n R D)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:qualifiedCardinality n. _:x owl:onClass D.

Language Feature	Functional Syntax	RDF Syntax
maximum cardinality	DataMaxCardinality(n R)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:maxCardinality n.
qualified maximum cardinality	DataMaxCardinality(n R D)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:maxQualifiedCardinality n. _:x owl:onClass D.
minimum cardinality	DataMinCardinality(n R)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:minCardinality n.
qualified minimum cardinality	DataMinCardinality(n R D)	_:x rdf:type owl:Restriction. _:x owl:onProperty R. _:x owl:minQualifiedCardinality n. _:x owl:onClass D.

2.10 Object and data property expressions

Language Feature	Functional Syntax	RDF Syntax
named object property	PN	PN
universal object property	owl:topObjectProperty	owl:topObjectProperty
empty object property	owl:bottomObjectProperty	owl:bottomObjectProperty
inverse property	ObjectInverseOf(PN)	_:x owl:inverseOf PN

Language Feature	Functional Syntax	RDF Syntax
named data property	R	R
universal data property	owl:topDataProperty	owl:topDataProperty
empty data property	owl:bottomDataProperty	owl:bottomDataProperty

We cannot write the inverse of a data propery because the value of a data property is a literal and in this way we will have a literal in subject position in RDF representation and we cannot use values as subjects in the sematnics Web.

2.11 Class axioms

Remember that in ALC we don't have any role axioms.

Language Feature	Functional Syntax	RDF Syntax
subclass	SubClassOf(C_1 C_2)	C_1 rdfs:subClassOf C_2.
equivalent classes	EquivalentClasses(C_1 … C_n)	C_j owl:equivalentClass C_{j+1}. j=1…n-1
disjoint classes	DisjointClasses(C_1 C_2)	C_1 owl:disjointWith C_2.
pairwise disjoint classes	DisjointClasses(C_1 … C_n)	_:x rdf:type owl:AllDisjointClasses. _:x owl:members (C_1 … C_n).
disjoint union	DisjointUnionOf(CN C_1 … C_n)	CN owl:disjointUnionOf (C_1 … C_n).

2.12 Object property axioms

Language Feature	Functional Syntax	RDF Syntax
subproperty	SubObjectPropertyOf(P_1 P_2)	P_1 rdfs:subPropertyOf P_2.
property chain inclusion	SubObjectPropertyOf(ObjectPropertyChain(P_1 … P_n) P)	P owl:propertyChainAxiom (P_1 … P_n).
property domain	ObjectPropertyDomain(P C)	P rdfs:domain C.
property range	ObjectPropertyRange(P C)	P rdfs:range C.
equivalent properties	EquivalentObjectProperties(P_1 … P_n)	P_j owl:equivalentProperty P_{j+1}. j=1…n-1
disjoint properties	DisjointObjectProperties(P_1 P_2)	P_1 owl:propertyDisjointWith P_2.
pairwise disjoint properties	DisjointObjectProperties(P_1 … P_n)	_:x rdf:type owl:AllDisjointProperties. _:x owl:members (P_1 … P_n).
inverse properties	InverseObjectProperties(P_1 P_2)	P_1 owl:inverseOf P_2.
functional property	FunctionalObjectProperty(P)	P rdf:type owl:FunctionalProperty.
inverse functional property	InverseFunctionalObjectProperty(P)	P rdf:type owl:InverseFunctionalProperty.

Language Feature	Functional Syntax	RDF Syntax
reflexive property	ReflexiveObjectProperty(P)	P rdf:type owl:ReflexiveProperty.
irreflexive property	IrreflexiveObjectProperty(P)	P rdf:type owl:IrreflexiveProperty.
symmetric property	SymmetricObjectProperty(P)	P rdf:type owl:SymmetricProperty.
asymmetric property	AsymmetricObjectProperty(P)	P rdf:type owl:AsymmetricProperty.
transitive property	TransitiveObjectProperty(P)	P rdf:type owl:TransitiveProperty.

2.13 Data property axioms

Language Feature	Functional Syntax	RDF Syntax
subproperty	SubDataPropertyOf(R_1 R_2)	R_1 rdfs:subPropertyOf R_2.
property domain	DataPropertyDomain(R C)	R rdfs:domain C.
property range	DataPropertyRange(R D)	R rdfs:range D.
equivalent properties	EquivalentDataProperties(R_1 … R_n)	R_j owl:equivalentProperty R_{j+1}. j=1…n-1
disjoint properties	DisjointDataProperties(R_1 R_2)	R_1 owl:propertyDisjointWith R_2.
pairwise disjoint properties	DisjointDataProperties(R_1 … R_n)	_:x rdf:type owl:AllDisjointProperties. _:x owl:members (R_1 … R_n).
functional property	FunctionalDataProperty(R)	R rdf:type owl:FunctionalProperty.

2.14 Assertions (ABox statements)

So far we have seen TBox statements.

Language Feature	Functional Syntax	RDF Syntax
individual equality	SameIndividual(a_1 … a_n)	a_j owl:sameAs a_{j+1}. j=1…n-1
individual inequality	DifferentIndividuals(a_1 a_2)	a_1 owl:differentFrom a_2.
pairwise individual inequality	DifferentIndividuals(a_1 … a_n)	_:x rdf:type owl:AllDifferent. _:x owl:members (a_1 … a_n).
class assertion	ClassAssertion(C a)	a rdf:type C.
positive object property assertion	ObjectPropertyAssertion(PN a_1 a_2)	a_1 PN a_2.
positive data property assertion	DataPropertyAssertion(R a v)	a R v.
negative object property assertion	NegativeObjectPropertyAssertion(P a_1 a_2)	_:x rdf:type owl:NegativePropertyAssertion. _:x owl:sourceIndividual a_1. _:x owl:assertionProperty P. _:x owl:targetIndividual a_2.
negative data property assertion	NegativeDataPropertyAssertion(R a v)	_:x rdf:type owl:NegativePropertyAssertion. _:x owl:sourceIndividual a. _:x owl:assertionProperty R. _:x owl:targetValue v.

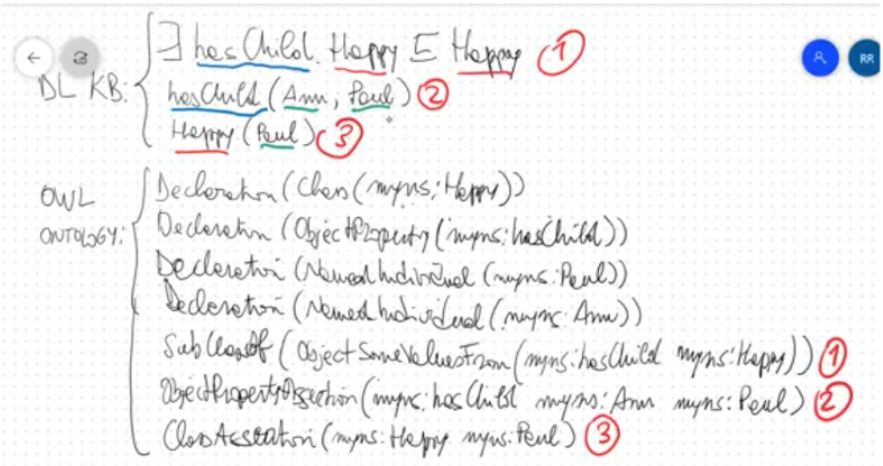
2.15 From a simple DL knowledge base to a OWL ontology representation

⇒ Concepts become Classes in OWL
⇒ Role become Object and Data Properties

EXAMPLE:
"If someone has a child that is happy, then he is happy".

First of all we have to write declarations: which are the classes, the properties, the object properties, the data properties and the individuals (see the Declaration tables). And then we write declarations for all the classes, the obeject properties, the data properties and the individuals.
There is only one class, Happy, and one role, hasChild. Ann and Paul are abstract individuals (not values because you cannot have literals in subject position, only URIs can appear.. and you know the subject of a property assertion becames the subject of an RDF triple in the RDF representation). The symbols must be: URIs, blank nodes or literals.
We have to understand if the role hasChild is a object or a data property: it is a object property because it relates inviduals, Ann and Paul, that are abstract objects, not value, so we must have URIs for them. Also concepts and roles must be URIs (Happy and hasChild), because for example the object properties are used in predicate position, and we know that in RDF only URIs can appear in predicate position.
→ All symbols, except data values, must became URIs.
- Do all the declarations (adding the prefix because they're URIs, in this case I invent my own URI: myns)
- 1) Let's move to the statement: write the concept inclusion as a subClassOf. The argument on the right is just a concept name, instead the one on the left is using a concept constructor that qualifis the existential restriction (The constructor in OWL to write an existential restriction is ObjectSomeValueFrom and it takes as first argument the property name hasChild and the second argument is the class which it qualifies, Happy).
- 2) Object property assertion: ObjectPropertyAssertion that takes as first argument the property name hasChild, as second and third arguments the URIs of Paul and Ann.
- 3) Class assertion: ClassAssertion that takes as first argument the class name Happy, as second argument the URI of Paul.

→ So this is the OWL syntax for the DL knowledge base of the example.
The functional syntax is standard infact almost all the tools accept input ontologies writtten in the functional syntax and for the moment it is the closest to the DL syntax.
At the end it can be translated both in RDF, producing a RDF grpah, and in this why we can write proper SPARQL queries →Queries on OWL are actually done using a language that refers to the RDF syntax, that is SPARQL language.



2.16 XML Schema datatypes in OWL

OWL supports XML Schema primitives datatypes, like integer, real, string.
As we said before, it creates a strict separation between "object" classes and datatypes.

2.17 OWL DL semantics

Remember that the semantics of DL we have studied was defined based on the concept of interpretation, that is pair domain of interpretation and interpretation function.

DL semantics defined by interpretations: $\mathcal{I}$- $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where

- $\Delta^{\mathcal{I}}$ is the domain (a non-empty set)
- $\cdot^{\mathcal{I}}$ is an interpretation function that maps:
  - Concept (class) name A $\rightarrow$ subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$
  - Role (property) name R $\rightarrow$ binary relation $R^{\mathcal{I}}$ over $\Delta^{\mathcal{I}}$
  - Individual name i $\rightarrow$ $i^{\mathcal{I}}$ element of $\Delta^{\mathcal{I}}$

But the DL we use for OWL is a more complicated one: SHOIN(D_n):

- Facilitates provision of reasoning services (using DL systems)
- Provides well defined and more expressive semantics
- It allows to use n datatypes: we need to extend the semantics because the datatype must no be interpreted in the abstract interpresation domain Δ^I, but in the real datatype domain. For example if we use integer we must add in our interpretation domain the integers; numbers are numbers, we cannot interprete the numbers, they have a predefined semantics and there are relation between them.
  The reasoner must be aware of the fact that there are real data values appearing in the ontology. (we do not have real data types in the origin DL approch)

2.18 OWL DL ontologies are DL knowledge bases

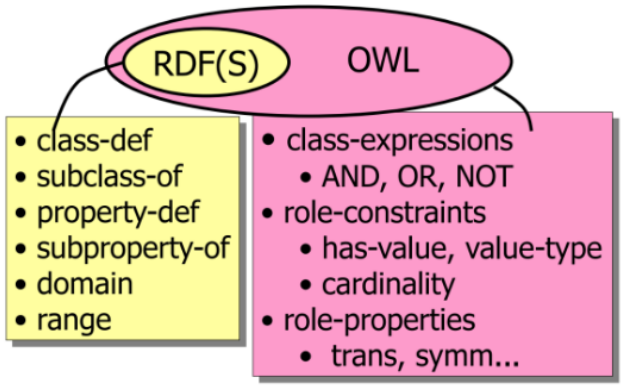
- An OWL ontology maps to a DL Knowledge Base

$\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$

  - $\mathcal{T}$(Tbox) is a set of axioms of the form:
    - $C \sqsubseteq D$ (concept inclusion)
    - $C \equiv D$ (concept equivalence)
    - $R \sqsubseteq S$ (role inclusion)
    - $R \equiv S$ (role equivalence)
    - $R^+ \sqsubseteq R$ (role transitivity)
  - $\mathcal{A}$(Abox) is a set of axioms of the form
    - $x \in D$ (concept instantiation)
    - $\langle x,y \rangle \in R$ (role instantiation)

2.19 OWL vs RDFS

All this stuff gives to OWL much more expressive power than RDFS. RDFS is just writing subClass, subProperty, domain and range relations between classes and properties. On the other hand, OWL allows for using all the boolean connectors (AND,OR,NOT) for writing class expressions, a lot of possible property restrictions (with number restrictions, functionality restrictions), reflexivity, simmetry and transitivity of properties, etc.



2.20 Inference tasks in OWL

Reasoning tasks in OWL are very similar to DL ones. Since OWL DL ontology is DL knowledge base, all the things that can be done on a DL KB should also be done on an OWL ontology.

- Ontology consistency (corresponds to KB consistency in DL)
- Concept/role consistency (same as DL)
- Concept/role subsumption and equivalence (same as DL)
- Instance checking (same as Dl)
- ...

But moreover, with OWL we also have **query answering**.
Until this moment we only talked about instance checking (the question I do to the KB is "tell me if this individual is an instance of this class or tell me if these two individuals are instances of a property) that asks very simple questions to our KB. These simple questions can be generalized and can become much more expressive using a proper query language, like SPARQL 1.1 (that includes the interpretation of SPARQL queries over an OWL ontology, but an OWL ontology represented by the corresponding RDF syntax, so it's an RDF graph).

But who is going to execute these reasoning tasks on an OWL ontology?
This is done by a reasoner thath uses DL algorithms for reasoning. At the moment the only algo for DL reasoning that we know is the tableau algo for ALC, but in general it can be extended for more expressive DL and also to OWL DL (but this extention was not so simple, the algo was published a coupe of years after the publication of the OWL standard itself).

2.21 SPARQL 1.1

SPARQL 1.1 became the query language for OWL ontology and it was developed by considered the three semantics problems we mentioned before: at the beginning SPARQL evalutation of a RDF graph was based on pure graph mapping, it means we evaluate query patterns over the graph looking for matches between the triple patterns (the basic graph pattern) in the SPARQL query and the real RDF graph, every match produces an answer; but in this way we are not giving any special semantics to the special labels that appears in the graph belonging to the RDF, RDFS and OWL vocabulary. We must give a special interpretation to these symbols, otherwise we are going to loose all the special meaning of these labels and, of corse, all the meaning of the ontology. We must define interpretation of SPARQL queries under semantics semantics, not just under graph match semantics. This was seriously addressed by the SPARQL 1.1 version, introducing several so called **entailment regimes**: it is a interpretation of some of the symbols appearing in the graph. The semantics of SPARQL queries for OWL is defined by two entailment regimes for SPARQL:
- **OWL 2 RDF-based semantics** entailed regime (used for OWL Full)
- **OWL 2 direct semantics** entailed regime (OWL DL - corresponds to DL semantics)

2.22 Computational aspects of reasoning

It was proved that, like with ALC, the complexity is exponential.
But as we said, reasoning in OWL-DL is decidable, even if the algo can terminate in exponential time.
Not only theoretical complexity (not only the theoretical worst case) is exponential, also practical complexity, reasoning cannot be performed in reasonable running time, expecially is we run these algos on medium-large size ontologies.

2.23 Current OWL technology

There are two kinds of tools for OWL:
- OWL editors
- OWL reasoners
The two things are merged together in the distribution: editors come with predefined reasoners embadded into the editors.

2.24 OWL editors

It allows for visualizing/browsing/editing OWL ontologies and it is able to connect to an external OWL reasoner, creating the OWL envoirment.
In the very early days of OWL there were editors, but there weren't reasoner: so you could use edit to create an ontology but then you did not have real algo that was able to analyze your ontology. This problem has slowly disappeared and after few years the performances became acceptable.

Main current tools:
- Protege (most famous)
- SWOOP
- OWLed2

2.24.1 OWL reasoning tools

two categories:

- OWL-DL reasoners, e.g.:
  - Hermit
  - Pellet
  - Konclude
  - Racer, RacerPro
  - Fact++

- reasoners for "tractable fragments" of OWL-DL, e.g.:
  - ELK (OWL 2 EL)
  - Mastro, Ontop (OWL 2 QL)
  - RDFox (OWL 2 RL)

Still today, only few of them support (or partially support) query answering according to the SPARQL 1.1 standard, because the majority of these technologies was developed before the adoption of the SPARQL 1.1 standard.

3 OWL2

OWL 2 is the second release of OWL. With first release in 2004, the semantic web community was *complaining* about the **performance** of OWL reasoner. The technologies for reasoning in this *ontology specification* was **extremely heavy** because the OWL-DL language **didn't have** property reasoner, algorithm for basic reasoning tasks.
Only OWL-LITE was equipped with real algorithm and real implementation of reasoner.
After some years, also the OWL-DL language was covered by next generation algorithms for this **ontology languages**. but there was a *big problem* related to the **performance**, implementation.

- OWL 1 = first release of OWL (2004)
- Three versions of OWL 1:
  - OWL Full: undecidable
  - OWL-DL: reasoning is exponential
  - OWL-Lite: almost same complexity as OWL-DL
- Main criticism: processing OWL is computationally too expensive (exponential)
- especially in Semantic Web applications, scalability (or at least tractability) of processing/reasoning is a crucial property

The original stardard, OWL1, was present in 3 different versions: *OWL-Full*, *OWL-DL* and *OWL-LITE*.
OWL-Full is the real extension of the *all RDF language* with RDF schema and *ontology part*. But it is prove to be **undecidable**. So, the real reason behind this negative property of the full *integration* of RDF and OWL is due to the **different semantics nature of RDF and OWL**. RDF has *meta-modeling ability* which are **not** originally included in description logic. So, creating this integration gave right some aspects that we are completely *unknow* at the beginning which lead to *undecidable* for some reasoning.
This **meta-modeling** ability, which is full integrated with full description logic ability lead to *undecidability*. THe language is very *expressive* but it is also *impossible* to find the **terminating** algorithm for *reasoning*, in this full integration of OWL and RDF.

In the other version, discussed in for OWL1, the RDF syntax was there but **not** the RDF semantic. The *meta-modeling* ability, the fact that the same symbol can be used to represent different objects and also predicates or individuals, was **not fully** supported by **OWL-DL** and **OWL-LITE**.
But the **main criticism** was related to the fact that the algorithms are *exponentials* but also in practical case, in which they are **not** consider very complicated (even small size of ontology), this reasoner were showing *extremely poor performance*. They were **not** able to manage *small ontologies* and this was really criticised.
In the Semantic web application, when you called information from different web sources, the **scalabilty** property, which means the fact that you should be able to handle very large instances (input files), is very important. This was **not completely** supported by this two tools.
In the Semantic web we need *scalable algorithm*, algorithms who's performance time is *linearly proportional* to the input size. They should be really good in performance, they *couldn't* even be *polynomial or quadratic*, it is already too much.
We have algorithms that are *exponentials* and this exponentiality was really showing up very early.
You can **get stuck** very frequently, even with small input size.

- performance of OWL-DL reasoners:
- "practically good" for the intensional level
  - the size of a TBox is not likely to scale up too much
- not good for the extensional level
  - unable to handle instances (ABoxes) of large size (or even medium size)...
  - ...even for the basic extensional service (instance checking)

The real difficulty of the **reasoners** in the early years of OWL was localted *mostly* in the ABox.
When the ontology has *large Abox*, a lot of instances and realatively *small TBox*, this was the scenario that has really problems in **performance**. This is **not** a surprise because if we consider also *tableau algorithm* it woks better with **small ABoxes** because when we apply the *existential rule* we are forcing to add a lot of formulas to the tableau. Every individual has a *big cost* because then we have to start again with very complex formula and decomposite with and/or rules etc until we get atomic formula.
Even very small knowledge base ontology we create this big problem that we have to many individuals and so the performance increase too much. These algorithms are **not able** to *terminate* in **accetable running time**.

This application in Semantic Web of OWL need *certainly* queries, so more sofisticated access methods in the ontology. We **would** like to have something like *query language*, some declarative way to specify *formal requests* that have sofisticated and expressive power. This was also another **limitation** because those reasoners were **only able** to run *consistency checks, subsumption checks, instance checking checks.* So, the *classical reasoning problems* of description logic. There were **not sofisticated** query facilities for the usage of this reasoners.

- why are these tools so bad with (large) ABoxes?
- two main reasons:
- current algorithms are mainly derived by algorithms defined for purely intensional tasks
  - no real optimization for ABox services
- these algorithms work in main memory
  => bottleneck for very large instances

The description logic technologies at the time **didn't** consider *seriously* the presence of the ABox. There is also an *historical reason* for this. All the **initial projects** that driving the standard, using just TBoxes and **not** ABoxes. The Abox problem was really **underestimated**, so this reasoner were **never** been tested with large ABoxes. **Only** after the standard appears we will see the limits of the technology at the time.
There was also problem with *memory*. The all knowledge base is parsed and stored in main memory as *object of main memory* and then all the processing is inside the main memory. It is good for the performance but there is a limitation in the *size of the main memory* so when you try to really work with input ontology that are very large it might **not** be possible to store every statement, every aspect of the ontology in main memory at the same time, *all together* because of the **phisical limit** of the main memory. Consider that **also** the algorithm consume al lot of memory.

Before the OWL standard recomentation, **no one** actually in the *description logic* community has notice that **size problem** could really happened in practise because they **didn't** plan a usage of description logic.
The technology was **not** prepared for this kind of massive usage with large dataset, with real web sources and the possibility of running dataset with hundred thousand of individuals.

- the limits of OWL-DL reasoners make it impossible to use these tools for real data integration on the web
- web sources are likely to be data intensive sources
- e.g., relational databases accessed through a web interface
- on the other hand, data integration is **the** prominent (future) application for Semantic Web technology! [Berners-Lee et al., IEEE Intelligent Systems, May 2006]

In most applications of the Semantic Web, it is very common to think application where the *number of class* is very large but **not** too large, so in TBox you write **only** relevant terms and they **cannot be huge**. Instead, with ABox you can have too many individuals. It is usual to find very large ABoxes.

- how to overcome these limitations if we want to build data-intensive Semantic Web applications?
- solution 1 : do not reason over ontologies
- solution 2: limit the expressive power of the ontology language

  => tractable fragments of OWL (**OWL profiles**)
- solution 3: wait for more efficient OWL-DL reasoners
- to arrive at solution 2, we may benefit from the new technology developed for OWL tractable fragments

What happened after this criticism?
First of all, there was a *reaction in standard*, study a second version of OWL. The idea was to *answer* or *reply* to the criticism by **creating special version of OWL** which lead *less expressive power* than the original OWL language.
We can solve the **performance problem** by creating versions of OWL that are easier to compute in terms of reasoning, reduce the **expressive power of the language**. Less the expressive power is, the easier is the reasoning of this language.
Let's try to create **OWL profiles**, fragments of OWL or sublanguages of OWL which are *tractable*. Tractable means that the reasoning in this language **should not be** *exponenial*, should be just **at most** *polynomial* w.r.t size of initial ontology. So, tractable means more or less easy to compute in terms of reasoning.

We have 3 possibles answers to the *criticism* about the complexity of reasoning in OWL. **One solution** is that we can use OWL but we **don't** run reasoning algorithm on OWL, this looks a *crazy idea*, but it is **not completely** crazy idea.
The idea of using OWL **not** for reasoning but just for documentation, which means that it is interesting to write an ontology even if you **don't** run *any automatic check on this ontology* might makes sense in some context in witch you just want to write a good formal documentation of some domain. In this way you are going to *miss all the important automatic checks* that you can run on the ontology (**correctness**). But you already get something from the language, in the sense that, you now have a language that *allows you to write* **abstract conceptual specification** of your domain even if you **don't run** any algorithm.
This is very *limited usage of OWL* because once you have language with formal *semantics* and in teory you have the possibility of running an algorithm that does automatically **correctness checks** for you, it is very disappointing **not** to use this possibility. This solution is quite limiting.
Second solution is the one solution descrived before on the **OWL profiles**.
Third solution says that the problem that we have found in the original version of OWL was due to the fact that technologies for OWL was at too early stage. It taks some years to have an *optimizer version of these algorithms*. The idea is to **wait** some years to have mush better implementation reasoners for OWL.

3.1 OWL profiles

- idea: sacrifice part of the expressiveness of the ontology language to have more efficient ontology tools
- OWL Lite is a standardized fragment of OWL-DL
- is OWL Lite OK?
- NO! it is still too expressive for ABox reasoning (OWL Lite is not really "lite"!)

The idea of *dicreasing the expressive power* of the language was **already** present in the first release of OWL, in the third version: **OWL-LITE**.
OWL-LITE is further fragment of OWL-DL. THe different between OWL-Lite and OWL-DL lies in a couple of *concep constructors*. At the time of recommendation there was **no** real implementation of reasoner of OWL-DL, the **only** available implementations was for OWL-LITE ontologies. So, those *two or three constructors* which were **excluded** by OWL-LITE were excluded just because the algorithm, existsing at the time, **could not process** these constructors.

OWL-LITE is **still** an *exponential language*, in fact, the reasoner for OWL-LITE has very poor performance. So, **OWL-LITE** was **not design** and **didn't** really address the *problem of performance*, it was just addressing the *problem of there exist implementation for reasoner for this language* but this implementation is **not efficient**.

- The second version of OWL (called OWL2) became a W3C recommendation on October 2009
- Besides the OWL2 Full language and the OWL2 DL language, this recommendation contains three fragments of OWL2 DL called **OWL 2 PROFILES:**
  - **OWL 2 QL** based on the DL **DL-Lite**
  - **OWL 2 EL** based on the DL **EL**
  - **OWL 2 RL** based on the DL **RL**

The *second release* of OWL decided to create **profiles**, sublanguages which were really light, in the sense that for thie languages it was really possible to define *efficient reasoner*, that was the idea.
These *profiles* actually are **three**: *OWL2 QL, OWL2 EL* and *OWL2 RL*.
OWL QL is the *fragment* oriented to queries.
OWL EL is the *fragment* oriented to existential restrictions and the conjuction.
OWL RL is the *fragment* that it is a kind of intersection between description logic and datalog rules.

Fragments means that the **syntax** is the same as OWL but a lot of class constructors, class axioms, property constructors or properties axioms are **forbidden** in this *profiles*.
In every profiles there is a *list of allowed* synttactic aspects of OWL that are possible and *list of things that cannot be done*. It is **restriction of the syntax of OWL**. So, we have three different restrictions of syntax of OWL. Every such restricted language corresponds to *different description logic*.
For instance, the *OWL2 QL fragment profiles* based on description logics of **DL-Lite**.

- Each of the three OWL 2 profiles has **tractable** complexity of reasoning: i.e., the main reasoning tasks (ontology consistency, instance checking, etc.) can be computed in polynomial time
- These polynomial reasoning techniques are very different from the general tableau algorithm for Description Logics (ALC)
- In particular, for query answering:
  - OWL 2 QL/DL-Lite: algorithm based on **query rewriting**
  - OWL 2 RL: algorithm based on **ABox materialization**
  - OWL 2 EL: algorithm based on a combination of query rewriting and ABox materialization

There is **common characteristic** for the *three profiles*: every such profiles has a **tractable complexity of reasoning**. **What does it mean?**
All the classical reasoning tasks (ontology consistency, instance checking etc) can **all be computed** in *polynomial time*. This is very different from ALC and other description logics because all of these tasks are*exponentials*. **How are the algorithms?**
The algorithms are **not the tableau algorithms**, but they are very different from tableau.
Also the techniques are **different** for the different *profiles*. This means that algorithm for reasoning in OWL2 QL profiles is based on technique called **query rewriting** and it is different from the algorithm for OWL2 RL, that it is based on **ABox materialization** that it is consider as simplify version of the *tableau algorithm*.
For OWL2 EL the algorithm is a kind of **combination of query rewriting and ABox materialization**.

3.1.1 OWL2 QL/DL-Lite

- DL-Lite is a tractable OWL-DL fragment
- defined by the DIS-Sapienza DASI research group
- main objectives:
  - allow for very efficient treatment of large ABoxes...
  - ...even for very expressive queries (conjunctive queries)

DL Lite is consider the basis OW2 DL standard.
What's the idea of this description logic?
To allow for very efficient treatement of large ABoxes. The very first description logic who's goal was to be able to have reasoning algorithm that **didn't get stuck** with *very large ABoxes* and allow for *expressive queries* of such dataset. So, **not only** instance checking but something that it is more complicated that instance checking, something similar to SQL queries.
The *final idea* makes description logics **closer** to databases, in the sense that databases are able to deal with tables and databases are able to handle complex queries. So, this description logic should be able to handle with complex queries. Of course, we **didn't** get exactly the same queries language as databases.

The language we consider is a *conjunctive queries*, it is the fragment of SQL that corresponds to SELECT, FROM, JOIN queries. PROJECT and UNION operators are possible, what is left is the NEGATION operator. That is the idea which changed completely the viewpoint of description logic. Description logic were mostly looking at TBoxes **not** looking at the ABoxes. This was revolutional because this approach said that we have to look at the ABoxes more than the TBoxes and we have to look for very expressive instance queries. Try to overcome the limitation and the criticism that we have done after the adoption of first version of OWL.

- DL-Lite is a family of Description Logics
- **DL-Lite$_{core}$** = basic DL-Lite language
- main DL-Lite dialects:
  - **DL-Lite$_F$** (DL-Lite$_{core}$ + role functionality)
  - **DL-Lite$_R$** (DL-Lite$_{core}$ + role hierarchies)
  - **DL-Lite$_A$** (DL-Lite$_F$ + DL-Lite$_R$ + attributes + domains)
- the current OWL 2 QL proposal is based on DL-Lite$_R$

DL-Lite is **not** the name of description logic but is the *family of description logics*. So, there are different versions of DL-Lite.
The OWL2 QL proposal is based on description logic called $DL-Lite_R$. different version of DL-Lite are due to the fact that can or cannot write some of the axioms in the different languages. Small variations.

$DL-Lite_F$ syntax

concept expressions:
- atomic concept A
- role domain $\exists R$
- role range $\exists R^-$

role expressions:
- atomic role R
- inverse atomic role R^-

DL-Lite$_F$ **TBox** = set of
- concept inclusions
- concept disjointness assertions
- functional assertions (stating that a role is functional)

DL-Lite$_F$ **ABox** = set of ground atoms, i.e., assertions
- A(a) with A concept name
- R(a,b) with R role name

$DL-Lite_F$ is DL-Lite with *functional roles*.
We are beack to DL terminology, we are going to talk about *concepts* and *roles*, **not** talk about *classes* and *properties*.
Remember: when we talk about DL we talk about concepts and roles, when we talk about OWL we talk about classes and properties. But of course concepts are classes and roles are properties.
Anyway the *difference* is the fact that when we are in description logic we have *generic roles*, instead when we are in OWL we have *object properties* and *data properties* because we have also data value in OWL.

The **syntax** of $DL-Lite_F$ is extremely simple. **Concept expression** *allowed* are just 3: *name of concept*, *qualified existential restriction*, $\exists R$, because in this language we **cannot** write qualification, and then *existential restriction relative to the inverse role*, $\exists R^-$, the participation in the role is the inverse role, it means that we

are talking about range of the role.
Qualified existential restriction is a set of objects appearing in the first position in the interpretation of the role, white *existential restriction relative to the inverse role* is the set of elements appearing in the second position in the interpretation of the role R.
Notice that we **don't** have OR, AND, NEGATION, UNIVERSAL RESTRICTION.
For the **role expressions** we have: *atomic role* and *inverse atomic role*, R^-.

What is a TBox in $DL-Lite_F$?
A **set of concept inclusion** using concept expressions or **set of concept disjointness assertions** or **a set of functional assertions**. *Disjointess* means that concept expression is a subconcept of the negation of concept expression, essentially we can write 2 concept expressions are *disjoint*. For instance, two concept names like *Student* and *Worker* are disjoint concept. We can see that negation is present but **only** in *concept disjointness assetions*. *Functional assertions* means that the role is functional, this is an unqualified number of restriction, **at most one restriction**. For instance, we can say **at most** one "hasEdge" to say that everybody has at most one partecipation to the role "hasEdge".
But we can also have *inverse functional*, we can say that also the participation of the second position in the role is functional, so at most one time every object can appear *at most one* time in the second position of the role. For instance, "hasASocialSecurityNumber", it is functional because every person can have at most one Social Security Number, but it is also inverse functional because every Social Security Number can appera at most ones in this role because **only** one person can appear for that Security Number.

What is the ABox of $DL-Lite_F$?
It is just usual **concept assertions** with individual and concept name, A(a), or with role name, R(a,b).

TBox:	MALE $\sqsubseteq$ PERSON	concept inclusion
	FEMALE $\sqsubseteq$ PERSON	concept inclusion
	PERSON $\sqsubseteq \exists$hasFather	concept inclusion
	$\exists$hasFather$^- \sqsubseteq$ MALE	concept inclusion
	PERSON $\sqsubseteq \exists$hasMother	concept inclusion
	$\exists$hasMother$^- \sqsubseteq$ FEMALE	concept inclusion
	MALE $\sqsubseteq \neg$FEMALE	concept disjointness
	funct(hasMother)	role functionality
ABox:	MALE(Bob), MALE(Paul), FEMALE(Ann), hasFather(Paul,Ann), hasMother(Mary,Paul)	

A possible $DL-Lite_F$ TBox and ABox.
In TBox and ABox, every term appear with *uppercase* is a **concept**, instead, with *lowercase* is a **role**.
funct(hasMother) is statement this is to asserts that "hasMother" is a **functional role**. It means that every object can partecipate *at most once* in the **domain position** of hasMother. Everybody has at most one mother.
All the axioms in the TBox **do not have** any boolean costructors, the **only negation** is the *concept disjointness*, (MALE $\sqsubseteq \neg$ FEMALE).
The $\exists$ is **non qualified** but we have the *inverse role*.

main expressive limitations of DL-Lite w.r.t. OWL-DL:

1. **restricted disjunction:**
   - no explicit disjunction
   - binary Horn implications (concept and role inclusions)
2. **restricted negation:**
   - no explicit negation
   - concept (and role) disjointness
3. **restricted existential quantification:**
   - e.g., no qualified existential concepts
4. **limited role cardinality restrictions:**
   - only role functionality allowed
   - not a "real" problem

There are a lot of **limitations**, in fact, respect to OWL-DL we have a lot of limitations:

- **No disjunctions**
- **No negation**, a part from the disjointness
- **Restricted existential quantification**, no qualified existential concept
- **Very limited usage of number restrictions**, we have **only** the role functionality. We **cannot** write the number restrictions in *combination with other* concept expressions. This is different from **OWL**. In OWL we have number of restrictions, at least and at most, that can be used together with others concept constructors, so we can write much more complicated statements about number restrictions. But this is **not a real problem** because there are a lot of domain where the most important *cardinality* restriction is that the **at most one restriction**. So, it is a problem, but it is **not** that common problem.

We have see that **w.r.t** *OWL-DL* we have a lot of limitations.

What about DL-Lite comparison with RDF/RDFS?

DL-Lite captures RDFS...

- RDFS classes = concepts
- RDFS properties = roles
- rdfs:subClassOf = concept inclusion
- rdfs:subPropertyOf = role inclusion
- rdfs:domain = role domain
- rdfs:range = role range

but: DL-Lite does not allow for meta-predicates

DL-Lite extends RDFS:

- "exact" role domain and range
- concept and role disjointness
- inverse roles
- functional roles

It is important a comparison with RDF schema because it is our *lower bound*, because we have already in RDF layer an *ontology language*, which is **RDF schema**. But Semantic Web need a *more expressive ontology language* than RDF schema. That's why we create an OWL layer, which is on top of RDF layer.
DL-Lite is a *generalization of RDF schema* because we can write the subClassOf, subPropertyOf, domain and range restrictions, just like in RDF schema. but there is still *some differences*. *DL-Lite* **doesn't capture** *meta-modeling ability* of RDFS. This because DL-Lite is a description logic, which is a *classical logic* and in the classical logic the alphabet of concept is **disjoint** from the alphabet of roles, which is also disjpoint from the alphabet of individuals. So, we **do not** have this *polymorphic usage* of the identifier.

DL-Lite also *extends* RDF schema, actually the way in which we write the *domain and range* in DL-Lite is *more powerful* than in RDF schema because RDFS **doesn't** really allow for talking about the projection of a property in the first or in the second position. While the $\exists R$ and $\exists R^-$ of DL-Lite are the exact projection of the property, which means that DL-Lite talks the exact *domain* and *range* of roles, instead, in RDF schema you can **only** type the domain and range with some other class.
Then we can say the fact that MALE and FEMALE are **disjoint** classes. This is something that you **cannot** do in RDF schema because you **don't have** any form of NEGATION. Everything is *consistent*, there is **no** possibility of writing negation.
The **functional role restriction** is something that RDF schema **doesn't allow** to express because functional restrictions are *negative statements*. Functional restrictions have *implicit negation* because you **cannot have** more than one.
Also the **inverse roles** are **not** allow.

- DL-Lite captures a very large subset of the constructs of conceptual data modeling languages (UML class diagrams, E-R)
- e.g., DL-Lite_A captures almost all the E-R model:
  - entities = concepts
  - binary relationships = roles
  - entity attributes = concept attributes
  - relationship attributes = role attributes
  - cardinality constraints (0,1) = concept inclusions and role functionalities
  - ...

⇒ DL-Lite = a simple yet powerful ontology language

Another *comparison* is that, the DL-Lite was also inspired by working of **conceptual data modeling**.
Conceptual modeling is **not** Semantic Web, it is something more related to databases.
There are conceptual modeling languages in information systems and databases or UML class diagrams. This modeling languages are considered *conceptual data modeling languages* and they talks about Classes, but also E-R schema create different thisngs that DL-Lite capture.
We can see that there is a **connection** between *conceptual modeling languages* and DL-Lite. This is important.
Why?
Conceptual models **did not** appear in computer science with the Semantic Web, they were already there.

What is the expressive power of DL-Lite w.r.t languages for conceptual modeling, that already existed, in the Semantic Web?
DL-Lite is very close to these conceptual modeling languages. We are really capturing most of the aspects that we have considered relevant in previous conceptual modeling approaches. These approaches are used a lot. It means that we are capturing the most important *patterns* that are needed in conceptual modeling.
Even if DL-Lite appears *extremely restricted* w.r.t OWL or to DL, it looks like **sufficiently powerful to capture most of the important aspects** of conceptual modeling.

tractability of TBox reasoning:

- all TBox reasoning tasks in DL-Lite are tractable, i.e., solvable in polynomial time

tractability of ABox+TBox reasoning:

- instance checking and instance retrieval in DL-Lite are solvable in polynomial time
- conjunctive queries over DL-Lite ontologies can be answered in polynomial time (actually in LogSpace) with respect to *data complexity* (i.e., the size of the ABox)

In the end, the proposal of DL-Lite that we could *get rid* of a lot of constructors and features of description logic, but still retaining some important modeling ability for creating a conceptual model, like an *ontology*.
After this, we studied **reasoning**. We will see that DL-Lite is **tractable**, so we can solve it in polynomial time the TBoxed reasoning tasks, also the *instance checking* etc.
But the **most important computational property** of DL-Lite is to *queries answering*, **queries evaluation**. This conjunctive queries fragment can be done in *polynomial time* w.r.t **size of the ABox**, data complexity.
Why do we care about the size of the ABox?
Because the size of the ABox is the *real factor* that we want to **scale with**. I consider in most usages of my algorithm the size of the ABox dominates the size of the queries and the size of the TBox.

The scenario that we want to have **optimal** is to consider very *large ABox*, relatively small TBox and query size. The DL-Lite is *polynomial* w.r.t size of the ABox.

a glimpse on the query answering algorithm:

- **query answering in DL-Lite can be reduced to evaluation of an SQL query over a relational database** (this is the **first-order rewritability** property)
- query answering by query rewriting + relational database evaluation:
  1. the ABox is stored in a relational database (set of unary and binary tables)
  2. the conjunctive query Q is rewritten with respect to the TBox, obtaining an SQL query Q'
  3. query Q' is passed to the DBMS which returns the answers

In practise, we have also defined an algorithm for **queries answering** in DL-Lite that is based on the idea that **query answering in DL-Lite can be reduced to evaluation of an SQL query over a relational database**. This idea is both theoretical and practical. It is practical because the idea is that we **do not want** to create an algorithm that start from scratch and creates a new technique to answering queries. Since our query language is a *fragment of SQL*, we want to translate our query answering problem into database query answering problem. This is *another link* from DL-Lite to database. The **solution** is obtained by exploiting on databases technologies.

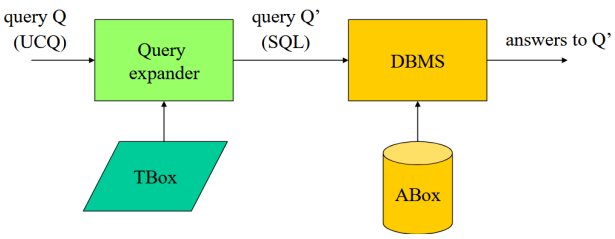
How do we write a Reasoner for DL-Lie?
We write an algorithm in which follows these steps:

1. The Abox is **stored in a relational database**. *For every class/concept* we create a table with one column and we store the instances of these concepts in this table. *For every role/property* we create a binary tables and we store all the instances of this role in the corresponding table. Represent ABox in the database table. Schema of database are concepts and roles of the knowledge base.

2. The we have a **query**, Q, and **TBox**. The are handled by an algorithm. This algorithm takes the query and the TBox and perform a **rewriting step**. Starting from conjunctive query and the TBox it produces another a new *SQL query, Q'*. This query Q' has a **semantic property** that it is exploited by the step 1).

3. Q' is **evaluated** on the databases computed at step 1). So, the answers produced by the Q' on the table stored in step 1) are the answers that, according to the semantic of description logic, should be produced by the original query Q on the original ontology formed by TBox and the ABox.

Graphical representation of query answering in DL-Lite



UCQ = Union of conjunctive queries.
Remember that means SELECT, PROJECT, JOIN, UNION statement, **without** NEGATION.
TBox becomes the inputs, together with query Q, over an algorithms called *query expander*, it takes the Q and TBox and rewrite the query producing a new query Q'. Q' is passed to the relational database system, which has already the ABox stored there, and DBMS produce the answers. The answers are the answers of original queries. This is completely different stategies w.r.t tableau algorithm.

What happens in this way?
We see that the ABox, which was the **critical aspect** of description logic Reasoners in 2006/2007, is *manages* by database system. Database is really good for managing large tables. the management of the ABox is **all related to DBMS**, with optimization and other technologies.
On the other hand the TBox is handle by **intensional Reasoner**. Query expander is able to consider the semantic of the TBox, consider the query, and it encodes the **semantic of the TBox in the rewriting of the query**. The *meaning* of the TBox is encoded in the query, but it is still express in SQL, in SQL because it is a language understandable by DBMS.
We are *splitting KB* in two. There is an algorithm for processing the TBox and there is a system for managing the ABox.

Example

TBox:
MALE $\sqsubseteq$ PERSON
MALE $\sqsubseteq \neg$FEMALE
$\exists$hasFather^{-} $\sqsubseteq$ MALE
$\exists$hasMother^{-} $\sqsubseteq$ FEMALE
FEMALE $\sqsubseteq$ PERSON
PERSON $\sqsubseteq \exists$hasFather
PERSON $\sqsubseteq \exists$hasMother

input query:
q(x) :- PERSON(x)

rewritten query:
q'(x) :- PERSON(x) $\vee$
FEMALE(x) $\vee$
MALE(x) $\vee$
hasFather(y,x) $\vee$
hasMother(y,x)

Quey is *give me the name of all the people.*
What the **query expanded** produced is the rewritten query (green block). It is **not** written in SQL but in logical, for simplicity but in SQL we have to use operator UNION.

rewritten query:
q'(x) :- PERSON(x) $\vee$
FEMALE(x) $\vee$
MALE(x) $\vee$
hasFather(y,x) $\vee$
hasMother(y,x)

ABox:
MALE(Bob)
MALE(Paul)
FEMALE(Ann)
hasFather(Ann,Paul)
hasMother(Paul,Mary)

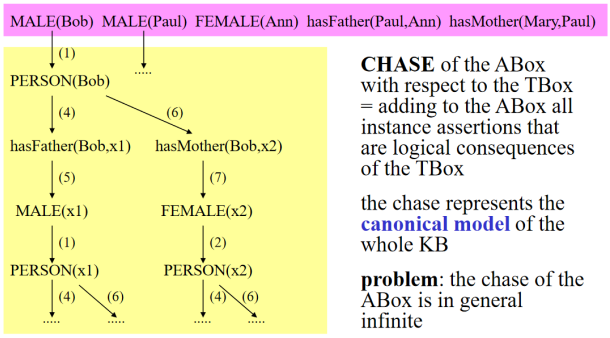
We run the rewritten query on the ABox and return the answers.

answers to query:
{ Bob, Paul, Ann, Mary }

Notice that if we have executed the original query on the ABox, the *answer* would be **empty** because there are **not** expliciting PERSON. This because the rewriting consider the meaning of the TBox w.r.t meaning of the query.
Remember: in the tableau every individual in the ABox must be instantiated with *all the formula* in the

TBox, in the GCI concept.
DL-Lite, which is DL that, it is the *basis* of the three OWL2 **profiles**, that is fragment or sublanguages of OWL2.
These *profiles* are intended to be more *efficients in terms of reasoning* w.r.t original OWL language because the **computational complexity of reasoning** in the OWL1 is very high and many people complained about this in the Semantic Web community.
DL-Lite is description logic behind OWL2-QL standard and we have seen that DL-Lite *admits* very particular reasoning technique. The *most important reasoning task* in DL-Lite is **query answering**. So, there are queries like SPARQL-query or SQL-query (where we can write conjunction of atoms or SELECT, FROM, WHERE statements or basic graph patter), in which we are able to perform JOIN operation, SELECT to specific value, PROJECT and also UNION.
These kind of query is evaluated over the *description logic knowledge base*, OWL2-QL ontology, and the way the query is answered is by **spitting TBOx and the ABox**, *intensional and extensional part of the knowledge base of the ontology*, and process the TBox w.r.t query Q and then produced a **rewriting of the query** Q'. This Q' is an *SQL-query* and it is passed to DBMS which actually managing the ABox (ABox is stored as a database). Q' is evaluated on the ABox and then produce the answers. In this final step the TBox is **discarded**, TBox is used **only to expand** the query Q w.r.t TBox itself using the *query expander*.
So, there is a *separation* between TBox and ABox that **improves** very much the efficiency of reasoning, in the sense that this technique is able to handle *very lange ABoxes*. This is very different from what standard DL reasoning algorithm are able to do. For example, the Tableau method is **not optimized** for instaces, many individuals in the ABox.

3.2 How is the query rewriting step done?



We can say that to answer queries the problem is that we have an ABox, which is essentially a database, but in addition to the ABox there is a TBox. THe Tbox statements allow for deriving ABox assetions that are **not written explicity in the ABox**. We can derive implicit knowledge from the Tbox and the ABox.
How can we address for the presence of the Tbox?
There are at least 2 ways:

- **First one**: *CHASING the ABox*.

CHASING the ABox is expand the ABox according to the axioms in the TBox.
In the picture we start with the ABox, that it is the pink one, and when we expand w.r.t. TBox we derive *new ABox assertions* starting from the previous assertions in the ABox and some TBox axioms.
The labels of the edges is exactly the number of axiom of the TBox used for that derivation, for example (1). This procedure is called **chase rule**. See example in previous Lesson or ABOVE this.

We derive new *assertions* from existing assertions in the ABox usig the Tbox axioms. The yellow part of the example is the expansion of the ABox w.r.t TBox.
In the case of *DL-Lite* this expansion is a **forest**, in the sense that, every assertion depends **only on one previous assertion**. For instance, PERSON(Bob) only depends on MALE(Bob). We have a *single dependence* from one derivation. This single dependence is a **peculiarity** of DL-Lite because *other description logics*, OWL-RL or OWL-RL, **don't have** this single dependence, so it is **not** true. This is the fact that DL-Lite is *simpler* DL language w.r.t other description logics.
We can take care the presence of the TBox by expanding the ABox. Once we have expanded the ABox, after applying all the axioms to derive all the possible assertions that are implied, we can answer query by *discarding the TBox*.
This is one of the possible wasy of the usage of the TBox.

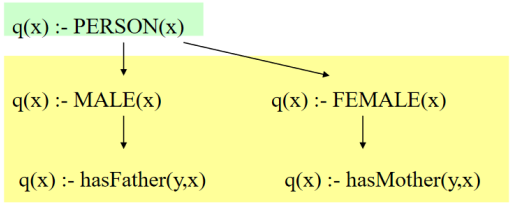
But there is a **problem** because this procedure to expand the ABox, which is called **CHASE**, in the case of DL-Lite produces an *infinite ABox*. The algorithm that implements the CHASE is **not** terminating in general.

Why?
Because, for example, from the fact MALE(Bob) we derive PERSON(Bob) but then there are axioms, PERSON $\sqsubseteq \exists$ hasFather and PERSON $\sqsubseteq \exists$ hasMother, that are called **mandatory participation to the role** (every person has a Father and every person has a Mother) which means that every person participate in role hasFather and every person participate in role hasFather in the domain position, first position. So, for this reason from PERSON(Bob) we derive that Bob participate in the role hasFather, hasFather(Bob, x1). x1 is the father but we **do not know** who is the father but there is a father for Bob. This is similar situation that we have seen in the *tableau* when we have the existential rule that says that you must add a new individual in the ABox. That exactly what we do here, *unknown individual*. We represent father with new individual name, for example x1. We do the same for Mother for x2.
From these two fact we have that hasFather belongs to the Class MALE and hasMother belongs to FEMALE. So, we derive other two formulas. We see that we derive from MALE(Bob) the MALE(x1). So, all the steps, CHASE steps, expansion steps that we have done for Bob we need to do also for x1. Essentially we create **infinite branches** with this tree, infinite chain of anchestors of Bob.
Here we have a *cyclic TBox*, but for **unfoldable TBox** we **doesn't have** this infinite problem.
This similarity between this CHASE and tableau for ALC is actually true. The CHASE is **nothing** that a very simplified form of tableau.
The *main difference* between CHASE and tableau for other description logic is the fact that the CHASE produce **only one ABox**. It is a kind of deterministic construction. While the tableau produces many ABoxes. But the principle between two methods is the *same* because we start from the ABox we apply the expansion rules. CHASE use directly TBox axioms to expand ABox formula instead of using GCI rule but the effect is more or less the same. In CHASE we **never** split because when we apply the CHASE we **do not have** OR operator. Because of this **infinite expansion** this is **not** a good algorithm to reason with DL-Lite knowledge base.

In fact, the *idea of DL-Lite* is **not to expand the ABox** but it is *expand the query*.

how to avoid the infinite chase of the ABox?

CHASE of the query:
- inclusions are applied "from right to left"
- this chase always terminates
- this chase is computed independently of the ABox



The CHASE operation is done on the *query* **not** on the ABox. **How do we do this?**
We do this in the *opposite direction* w.r.t TBox inclusions (concept inclusions). Inclusion are applied from right to left, the opposite that we have done for the ABox in which we apply inclusions from left to right. Inclusion in ABox are called **forward chaining**, we use the implication following the direction of implication. Instead, in the query rewriting we use **backward chaining**, we apply the rule in the opposite direction. For instance, from PERSON to MALE. From the *conclusion to the premises* of the formula, from consequence to preconditions.

Even though the rules are still cyclic, the cycle **doesn't happen** because the *backward chaining* of the rewriting rule applies **only** under some conditions of the variables of the query.
Cycles that we have seen in ABox are **broken** in the query rewriting because there are conditions in the application of the concept inclusions that **don't allow** for creating infinite rewriting. For example, in the picture above we stop in hasFather because the next step is that the variable y is a PERSON but since we are returning something about x, so we **cannot** do rewriting involving **only** a variable y and forgetting variable x that it is the real subject of our query.
There is a *technical limitation* that says that you **cannot** apply inclusion backward if they **don't consider** the relevant varaibles in this rewriting operation.
It is true that for every DL-Lite TBox and for every conjunctive query the rewriting of the query is **finite** because there are rules that preventing of creating inifinite expansion of the query.
ABox expansion can be *infinite* and query expansion can **never** be *infinite*. That is why DL-Lite is base on **query rewriting** instead of ABox expansion.

Essentially in the query expanded DL-Lite there are *two* basic **rewriting functions**: the first is *atom-rewrite*, every time you take a subclass of the previous atom you apply an atom rewrite, the second is *reduce*, when in

the query there are two atoms with the same predicate/role these two atoms can be **unified** so reduced to one atom. Reduce is *necessary* because otherwise the algorithm would be **not complete** w.r.t semantic of DL-Lite.

```
Algorithm PerfectRef (q, 𝒯)
Input: conjunctive query q, DL-Lite TBox 𝒯
Output: union of conjunctive queries PR
PR := {q};
repeat
    PR0 := PR;
    for each q ∈PR0 do
    (a) for each g in q do
            for each positive inclusion I  in 𝒯do
                if I is applicable to g
                then PR := PR ∪{atom-rewrite(q,g,I)};
    (b) for each g1, g2 in q  do
            if g1 and g2 unify then PR := PR ∪{reduce(q,g1,g2)}
until PR0 = PR;
return PR
```

The *rewriting* algorithm, that it is called PerfectRef, takes the initial query and then apply exhaustingly the *atom-rewrite* and the *reduce* step adding new conjunctive queries. So, we start with conjunctive query and we return with a set of conjunctive queries (it is just a union of conjunctive queries). Essentially, we return number of rewriting of the same query and all this rewriting are put into a union and that's the final rewriting of the query. This is done in exhausting way.
Since there is a theorem that say that you can **only** generate a finite number of rewriting for a query because of the fact that you **cannot** generate a cyclic expansion of the query, essentially, this algorithm **always terminate** and produced a correct answer.

- this query answering technique is in LOGSPACE with respect to data (ABox) complexity
- polynomial technique for deciding KB consistency in DL-Lite
- all main reasoning tasks in DL-Lite can be reduced to either KB consistency or query answering

  => all main reasoning tasks in DL-Lite are tractable

What about the complexity of query answering?
The complexity is actually very good because if we just consider the ABox size when we measure the complexity of reasoning of query answering. We talk about *data complexity*, we consider the ABox much larger than TBox and the query. Essentially, we ignore the size the TBox and the query when we measure the complexity.
If we **only** consider as a parameter of the complexity the size of the Abox, the technique is exactly like the evaluation of SQL query on the relation database. The complexity is **LOGSPACE**, similar to databases. The complexity of query ansewring in DL-Lite is similar to database, so, also for this reason we can do the *reduction* to database because they compatible. Also other reasoning tasks, **not only** query answering are *tractable*, polynomial.

OWL2-QL profile is much simpler than reasoning in OWL2-DL language, in the all language in OWL2. This profile is really more efficient w.r.t the complexity of reasoning.

3.3 Description Logic RL

concept expressions:
- atomic concept A
- concept conjunction $C_1 \sqcap C_2$
- qualified existential $\exists R.C$
- qualified existential $\exists R.\bot$

role expressions:
- atomic role R
- inverse role R^-

- RL **TBox** =
  - set of concept inclusions of the form $C \sqsubseteq A$ or $C \sqsubseteq \bot$
  - set of role inclusions $R_1 \sqsubseteq R_2$
- RL **ABox** = set of ground atoms, i.e., assertions
  - A(a) with A concept name
  - R(a,b) with R role name

What about the complexity of query answering?
In OWL2-RL profile "R" stands for *rule*. This fragment was created by the consortium, and it is a kind of **intersection** between *OWL2* and *datalog*. So, let's take the subclass of the language OWL2 that is expressible in datalog, that was the idea. First, there was the standard and then this description logic. What is interesting in this description logic is that, it can be express in **datalog**. There are *concept expression*: **atomic concept, conjunction, qualified existential restriction** and **existential qualified with bottom**. It looks a little bit closer to ALC because it has the conjunction operator and existential restriction qualification.
Here the we have the *qualified* existential restriction, instead in DL-Lite we **don't** have it but we have *unqualified* existential restriction, $\exists R.\top$ but **not** $\exists R.C$ with C that is a concept expression. For *role expression*:**atomic role and inverse role**.

What is a TBox in RL?
A TBox is a *set of concept inclusions*, but they have the form of **not** the general concept inclusion but $C \subseteq A$ or $C \subseteq \bot$. It means that in the right side of concept inclusion you **cannot** write an arbitrary expression because A is an *atomic concept* and $\bot$ is fixed.
In the TBox we can also have a *role inclusion* and this is similar to DL-Lite.

What is a ABox in RL?
ABox is a *set of ground atom* just like in DL-Lite.
Notice that we have generate a language that it is **different** from DL-Lite but then it has something less in the *concept inclusion* because in DL-Lite we are free to write whatever expressions in the right side, while in this RL language we can **only** write *concept names or* $\bot$.
Even though the concept expression are more *expressive*, the axioms are more *restricted*. We have something less w.r.t ALC because we **do not have** the NEGATION, OR, UNIVERSAL quantification but we have the inverse role.

TBox:

MALE $\sqsubseteq$ PERSON
FEMALE $\sqsubseteq$ PERSON
hasMother $\sqsubseteq$ hasParent
hasFather $\sqsubseteq$ hasParent
MALE $\sqcap$ FEMALE $\sqsubseteq \bot$
STUDENT $\sqcap$ EMPLOYEE $\sqsubseteq$ WORKING-STUDENT
$\exists$hasParent.HAPPY $\sqsubseteq$ HAPPY

ABox:

MALE(Bob), MALE(Paul), FEMALE(Ann),
hasFather(Paul,Ann), hasMother(Mary,Paul),
HAPPY(Ann), EMPLOYEE(Paul), STUDENT(Paul)

Here we can write the *disjointness* because in this case we have the AND and **not** the negative inclusion like in DL-Lite.
We can express also intersection, something that it is **not** possible to express in DL-Lite.

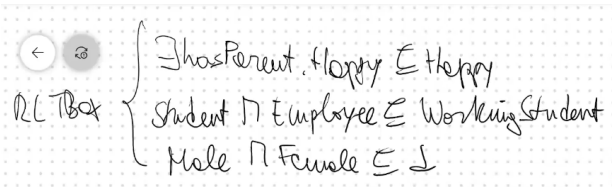
Complexity of reasoning in RL:

- Intensional (TBox) reasoning is PTIME-complete (i.e., tractable)
- Instance checking is PTIME-complete
- Conjunctive query answering is PTIME-complete with respect to data complexity
  - This implies that first-order rewritability does NOT hold for RL
- Conjunctive query answering is NP-complete with respect to combined complexity
- Reasoning in RL can be reduced to reasoning in positive Datalog

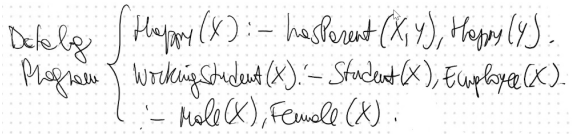
Reasoning in this *profile* is easier than in OWL, in fact it is true that the complexity is **PTIME**, so polynomial time, so tractable. Also instance checkig problem is solved in polynomial time.
Conjunctive query answering is **PTIME-complete** w.r.t *data complexity* but for DL-Lite is was *LOGSPACE*. So, in this case Conjunctive query answering is harder than Dl-Lite case.

In fact, this result is *partially positive* because it is **not exponential** but also *partially negative* because it says that the theoretical property called **first-order rewritability** for which it is possible to use the schema of query answering adopted by DL-Lite that separates TBox and ABox and delegates the ABox processing to the relational database is **not possible**.

Why?
The problem is that when you try to rewrite the conjunctive query w.r.t the RL TBox the query produced is **infinite**. So, there are cycles also in the *query rewriting process*. You have to do something different or you need to use a more expressive query language than SQL.
The final sense is that we **cannot** do the query rewriting in SQL just like in DL-Lite. But, reasoning in RL can be done by *translating the knowledge base* in **positive Datalog** program. This is **not only** a theoretical result but also practical because you are doing reasoning in RL by translating your KB into Datalog and then run datalog engine and in this way you are able to do reasoning in RL.



How do we translate this RL TBox in Datalog?
Remeber that implication are written in opposite direction in the Datalog For instance Happy(X) :- hasParent(X,Y).



How do we translate this RL TBox in Datalog?
Concepts are unary atoms, for instance Happy(X).
You can **always** translate every TBox axioms in RL and then translat these axioms in **positive Datalog rules** with constraints, see the third row.
If you want to reason there is a possibility because you translate your RL KB in Datalog and then run a reasoning algorithm, for example, Semi-naive or naive evaluation in Datalog.

Translation in Datalog is a possibility for **reasoning for RL** knowledge bases.
We know that *ground Datalog* programs **admitt polynomial time** reasoning algorithm. The problem is that when we have variables, with variables the time need to compute the *minimal model* can be **exponential**.
In our analysis we are talking about *data complexity*. So, we are **not** saying that the problem in the overall complexity is polynomial. Also in the case of Datalog programs obtained by translation from RL KB, you see that the predicates used in the translation are **only** concept and roles. So, the arity of every role is *fixed* can be **only one or two**. In Datalog, in general, this is **not true**, you can invent the arity, but in DL we have at most arity and at leat 1 in the predicate. This makes the difference w.r.t the complexity of reasoning.
This is the reason why the Datalog program, even if it has variables, when you compute the minimal model, it takes the **polynomial time** because if you fix the arity of the predicate then the complexity of reasoning Datalog is polynomial. The complexity is *exponential*, even with non ground program, **only** if you leave the user free to decide the arity of the predicates.
One way of solving the reasoning problem of RL is to translate the RL KB in Datalog and do reasoning in Datalog.

ABox reasoning and query answering in RL (and RDFS) can be done through **forward chaining** (a.k.a. **materialization**), which corresponds to the **chase** procedure mentioned above.

- Chase of the ABox with respect to the TBox = adding to the ABox all instance assertions that are logical consequences of the TBox
- In the case of RL (and RDFS) no new individual is introduced by the chase, so this procedure always terminates (and requires polynomial time)
- After this materialization step, the TBox can be discarded and conjunctive queries can be answered by evaluating them on the materialized ABox

In RL if you try to expand the query rewriting you will get an **infinite expansion**, instead if you try to expand the ABox you would get a **finite ABox**. It is exactly the opposite of DL-Lite.
In fact, the techniques that are *good* for DL-Lite are bad for RL, and viceversa. So, we go back to the CHASE of the ABox in this case of RL. Reasoning in RL usually it is done by a method that it is based on the **CHASE** procedure. THis technique is called **forward chaining**.

We use implication from left to right of the implication, so in forward direction, and it is also called **materialization** because we are materialize the implicit consequences of the TBox and ABox.
Tne ABox materialization can be done in *polynomial time*, quadratic time w.r.t initial size of the Abox. It is really relatively efficient.
When we translate RL KB into Datalog program and then we do a computation of the minimal model with semi-naive evaluation, what we do is exactly what is called **ABox materialization in description logic**, we generate the consequences of the rules until we reach the fix point. RL is a kind of intersection between OWL and Datalog, it is a fragment of OWL that can be expressed by positive Datalog rules.
In the end, the fact that, you have something that corresponds to Datalog is the reason why you can do Abox materialization in polynomial time, because it is just computation of the minimal model written in another way.

TBox:

MALE ⊑ PERSON
FEMALE ⊑ PERSON
hasMother ⊑ hasParent
hasFather ⊑ hasParent
MALE ⊓ FEMALE ⊑ ⊥
STUDENT ⊓ EMPLOYEE ⊑ WORKING-STUDENT
∃hasParent.HAPPY ⊑ HAPPY

ABox:

MALE(Bob), MALE(Paul),
FEMALE(Ann),
hasFather(Paul,Ann),
hasMother(Mary,Paul),
HAPPY(Ann), EMPLOYEE(Paul),
STUDENT(Paul)

TBox:

MALE ⊑ PERSON
FEMALE ⊑ PERSON
hasMother ⊑ hasParent
hasFather ⊑ hasParent
MALE ⊓ FEMALE ⊑ ⊥
STUDENT ⊓ EMPLOYEE ⊑ WORKING-STUDENT
∃hasParent.HAPPY ⊑ HAPPY

Materialized ABox (chase):

MALE(Bob), MALE(Paul),
FEMALE(Ann),
hasFather(Paul,Ann),
hasMother(Mary,Paul),
HAPPY(Ann), EMPLOYEE(Paul),
STUDENT(Paul),

PERSON(Bob), PERSON(Paul),
PERSON(Ann),
hasParent(Paul,Ann),
hasParent(Mary,Paul),
HAPPY(Paul), HAPPY(Mary),
WORKING-STUDENT(Paul)

We have to do *matrialization of the ABox* or you can call chase of the ABox or computational of the minimal model of the program corresponding into the translationin datalog of TBox axioms. You produce **always** the same things, these things are *new atoms*, slightly darker pink box, where we apply the **forward chase**.
We can also *answer query* because we can directly evaluate the query on this database, but before doing this we have to **expand the ABox**.

TBox:

MALE ⊑ PERSON
FEMALE ⊑ PERSON
hasMother ⊑ hasParent
hasFather ⊑ hasParent
MALE ⊓ FEMALE ⊑ ⊥
STUDENT ⊓ EMPLOYEE ⊑ WORKING-STUDENT
∃hasParent.HAPPY ⊑ HAPPY

Query: (happy grandchildren)

q(x) :- HAPPY(x), hasParent(x,y), hasParent(y,z).

Answer = { Mary }

Materialized ABox:

MALE(Bob), MALE(Paul),
FEMALE(Ann),
hasFather(Paul,Ann),
hasMother(Mary,Paul),
HAPPY(Ann), EMPLOYEE(Paul),
STUDENT(Paul),

PERSON(Bob), PERSON(Paul),
PERSON(Ann),
hasParent(Paul,Ann),
hasParent(Mary,Paul),
HAPPY(Paul), HAPPY(Mary),
WORKING-STUDENT(Paul)

Example of the query evaluation with this ABox.

RL is a real contact point between description logic and Datalog.

3.4 OWL2-EL

OWL2-EL is based on the description logic that is called EL. "E" stands for *existential*. So, it is the description logic based on the existential qualification and it is similar to RL but it is **not** translated into Datalog.
Statistically some researchers have seen that more than 90% of the things can be expressed **only** by existential qualification and conjunction. This is why EL comes in to play.

concept expressions:
- atomic concept A
- concept conjunction $C_1 \sqcap C_2$
- qualified existential $\exists R.C$

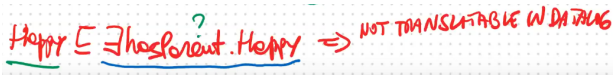
role expressions:
- atomic role R

- EL **TBox** = set of concept inclusions
- EL **ABox** = set of ground atoms, i.e., assertions
  - A(a) with A concept name
  - R(a,b) with R role name

Concept expressions: **names, concept conjunction** and **qualified existential restriction**. For *role expression* **only** the role and **not** the inverse. This is difference from DL-Lite and RL.
TBox is just a *set of concept inclusions*. be careful because Dl-Lite and RL have **also role inclusion**, here we **do not** have it.
ABox is *set of ground atoms*.
We have a language that it looks like similar to RL but it **doesn't have** the *inverse role* and *role inclusion*

There is one thing that can be done in EL and **not** in RL.
Concept inclusions are **general** in EL, so you can write whatever inclusion, while in RL we **cannot write arbitrary concept expression** in the superclass position.

Why do we want in RL only concept names or $\bot$ in a superclass position of inclusion?
Because otherwise we **cannot** translate axioms into Datalog.



I **cannot** write it in Datalog.

In EL we **do not care** about Datalog so we **cannot** have any restriction.

TBox:
MALE ⊑ PERSON
FEMALE ⊑ PERSON
PERSON ⊑ ∃hasFather.MALE
PERSON ⊑ ∃hasMother.FEMALE
STUDENT ⊓ EMPLOYEE ⊑ WORKING-STUDENT

Axiom 3 and 4 are **not** possible in RL.

ABox:
MALE(Bob), MALE(Paul), FEMALE(Ann),
hasFather(Paul,Ann), hasMother(Mary,Paul),
HAPPY(Ann), EMPLOYEE(Paul), STUDENT(Paul)

What about the computational properties?

Complexity of reasoning in EL (and in other languages of this family):

- Intensional (TBox) reasoning is PTIME-complete (i.e., tractable)
- Instance checking is PTIME-complete
- Conjunctive query answering is PTIME-complete with respect to data complexity
  - This implies that first-order rewritability does NOT hold for EL
- Conjunctive query answering is NP-complete with respect to combined complexity

It is pretty much the same of RL. The results are same of RL.
Also in El you **cannot** use the query rewriting schema of DL-Lite, but there is also another problem. You have problem of expanding ABox and also query because you have cycles. It seems that you can do **nothing**, but in reality you can think of making *smart combination of partial ABox materialization* and *partial query rewriting*. You can do a kind of **hybrid technique**, that give the same result of RL in the complexity.

In the end, EL is a different logic w.r.t RL because it is **not** Datalog translatable, it **doesn't** have the *finiteness of the ABox*, it also requires an *hybrid technique* between DL-Lite and RL approaches, but the computational properties of reasoning are the same of RL.

Reasoning technique	OWL 2 QL	OWL 2 RL	OWL 2 EL	OWL 2 DL
ABox materialization (forward chaining)	NO	YES	NO (*)	NO
Query rewriting (backward chaining)	YES	NO	NO (*)	NO
Tableau method	YES	YES	YES	YES

(*) A combination of partial ABox materialization and partial query rewriting can be applied to EL ontologies (see [Kontchakov et al., 2011])

This is the summarize of the OWL2 profiles.
The tableau method can address **all of these cases** but there is a problem with tableau because it is an *exponential* method.
If we look at the OWL2 DL, so all description logic fragment, which is a superset of ALC (it contains all the aspects), it **doesn't admitt** finite ABox and query rewriting but it admitts tableau.

For expressive description logic the most common method that it is used for reasoning is the *tableau method*. But for **restricted language**, like OWL profiles, there are specialized methods that are more efficient, polynomial, and work much faster.

Which is the best one?
We need to look at two aspect: *expressiveness* vs *performance* of reasoning.
If you choose a more expressiveness, like OWL-DL, you have the possibility of writing more precise description of you domain of interest but you will pay when you do the reasoning task because automated reasoning it's *exponential time*.
If you move to *polynomial* languages you will have better performance in reasoning but you have more restricted language so it is difficult to write a precise representation of you domain of interest, less expressive. This is the **trade-off**. It depends on you project, what you want to consider most. In particular, it depends on what's is the domain that you are going to model, what's the knowledge that you have to model and so on.