OWL 2
Web Ontology Language

Some material adapted from presentations by Ian Horrocks and by Feroz Farazi

Introduction

• OWL 2 extends OWL 1.1 and is backward compatible with it
• The new features of OWL 2 based on real applications, use cases and user experience
• Adopted as a W3C recommendation in December 2009
• All new features were justified by use cases and examples
• Most OWL software supports OWL now

Features and Rationale

• Syntactic sugar
• New constructs for properties
• Extended datatypes
• Punning
• Extended annotations
• Some innovations
• Minor features

Syntactic Sugar

• OWL 2 adds features that
  – Don’t change expressiveness, semantics, complexity
  – Makes some patterns easier to write
  – Allowing more efficient processing in reasoners
• New features include:
  – DisjointClasses
  – DisjointUnion
  – NegativeObjectPropertyAssertion
  – NegativeDataPropertyAssertion
Syntactic sugar: disJointClasses

- It’s common to want to assert that a set of classes are pairwise disjoint
  - No individual can be an instance of 2 of the classes in the set
- Faculty, staff and students are all disjoint
  - \[ \text{owl:allDisjointClasses; owl:members (:faculty :staff :students)} \]
- In OWL 1.1 we’d have to make three assertions
  - \[ (:\text{faculty} :\text{disjointWith :staff}) \]
  - \[ (:\text{faculty} :\text{disjointWith :student}) \]
  - \[ (:\text{staff} :\text{disjointWith :staff}) \]
- Will be cumbersome for large sets

Syntactic sugar: disJointUnion

- Need for disjointUnion construct
  - A \text{:CarDoor} is exclusively either
    - \[ (:\text{FrontDoor} :\text{RearDoor} :\text{TrunkDoor}) \]
    - and not more than one of them
- In OWL 2
  - \[ (:\text{CarDoor} :\text{owl:disjointUnionOf (:FrontDoor :RearDoor :TrunkDoor)}.) \]
- In OWL 1.1
  - \[ (:\text{CarDoor} :\text{owl:unionOf (:FrontDoor :RearDoor :TrunkDoor)}.) \]
  - \[ (:\text{FrontDoor} :\text{owl:disjointWith :ReadDoor} .) \]
  - \[ (:\text{FrontDoor} :\text{owl:disjointWith :TrunkDoor} .) \]
  - \[ (:\text{RearDoor} :\text{owl:disjointWith :TrunkDoor} .) \]

Syntactic sugar: negative assertions

- Asserts that a property doesn’t hold between two instances or between an instance and a literal
- NegativeObjectPropertyAssertion
  - Barack Obama was not born in Kenya
- NegativeDataPropertyAssertion
  - Barack Obama is not 60 years old
- Encoded using a “reification style”
Syntactic sugar: negative assertions

@prefix dbp: <http://dbpedia.org/resource/> .
@prefix dbpo: <http://dbpedia.org/ontology/> .
[a owl:NegativeObjectPropertyAssertion;
 owl:sourceIndividual dbp:Barack_Obama ;
 owl:assertionProperty dbpo:born_in ;
[a owl:NegativeDataPropertyAssertion;
 owl:sourceIndividual dbp:Barack_Obama ;
 owl:assertionProperty dbpo:age ;
 owl:targetIndividual "60"] .

• Note that the negative assertions are about individuals
• Suppose we want to say that :john has no spouse?
• Or to define the concept of an unmarried person?
• Can we use a negative assertion to do it?

Syntactic sugar: negative assertions

• Suppose we want to say that :john has no spouse?
  [a owl:NegativeObjectPropertyAssertion;
   owl:sourceIndividual :john ;
   owl:assertionProperty dbpo:spouse ;
   owl:targetIndividual ???????] .

Syntactic sugar: negative assertions

• The negative assertion feature is limited
• Can we define a concept :unmarriedPerson and assert that :john is an instance of this?
• We can do it this way:
  – An unmarried person is a kind of person
  – and a kind of thing with exactly 0 spouses
John is not married

:john a :unmarriedPerson.
:unmarriedPerson
a Person;
  a [a owl:Restriction;
    onProperty dbpo:spouse;
    owl:cardinality “0”] .

New property Features

• Self restriction
• Qualified cardinality restriction
• Object properties
• Disjoint properties
• Property chain
• keys

Self restriction

• Classes of objects that are related to themselves by a given property
• For example, the class of processes that regulate themselves
• It is also called local reflexivity
• An example: Auto-regulating processes regulate themselves
• narcissists are people who love themselves

Qualified cardinality restrictions

• Qualifies the instances to be counted
• Six varieties: {Data|Object}{Min|Exact|Max} Type
• Examples
  – People with exactly 3 children who are girls
  – People with at least 3 names
  – Each individual has at most 1 SSN
  – Pizzas with exactly four toppings all of which are cheeses
Qualified cardinality restrictions

- Done via new properties with domain owl:Restriction, namely \([\text{min}|\text{max}]\)QualifiedCardinality and \(\text{onClass}\)
- Example: people with exactly three children who are girls
  
  [a owl:restriction;
  owl:onProperty :has_child;
  owl:onClass [owl:subClassOf :FemalePerson;
  owl:subClassOf :Minor].

QualifiedCardinality “3”.

Object properties

- ReflexiveObjectProperty
  - Globally reflexive
  - Everything is part of itself
- IrreflexiveObjectProperty
  - Nothing can be a proper part of itself
- AsymmetricObjectProperty
  - If \(x\) is proper part of \(y\), then the opposite does not hold

Disjoint properties

- E.g: you can’t be both the parent of and child of the same person
- DisjointObjectProperties
  - Deals with object properties
  - Pairwise disjointness can be asserted
  - E.g., connectedTo and contiguousWith
- DisjointDataProperties
  - Deals with data properties
  - Pairwise disjointness can be asserted
  - E.g., startTime and endTime of a surgery

A Dissertation Committee

A dissertation committee has a candidate who must be a student and five members all of whom must be faculty. One member must be the advisor, another can be a co-advisor and two must be readers. The readers can not serve as advisor or co-advisor.
A Dissertation Committee

:DC a owl:Class; [a owl:Restriction;
  owl:onProperty :co-advisor; owl:maxCardinality “1”].
:candidate a owl:FunctionalProperty;
  rdfs:domain :DC; rdfs:range student.
:advisor a owl:FunctionalProperty;
  rdfs:domain :DC; rdfs:range faculty.
:co-advisor owl:ObjectProperty;
  rdfs:domain :DC; rdfs:range faculty,
  owl:propertyDisjointWith :advisor.
...

Property chain inclusion

• Properties can be defined as a composition of other properties
• The brother of your parent is your uncle
  :uncle owl:propertyChainAxiom (:parent :brother)

Keys

• Individuals can be identified uniquely
• Identification can be done using
  — A data property
  — An object property or
  — A set of properties
• Example
  foaf:Person owl:hasKey (foaf:mbox);
  owl:hasKey (:homePhone :foaf:name).

Extended datatypes

• Extra datatypes
  — Examples: owl:real, owl:rational, xsd:pattern
• Datatype restrictions
  — Range of datatypes
  — For example, adult has an age >= 18
    — DatatypeRestriction(xsd:integer minInclusive 18)
• Datatype definitions
  — New datatypes
    — DatatypeDefinition(adultAge DatatypeRestriction(xsd:integer minInclusive 18))
Extended datatypes

- Data range combinations
  - Intersection of
    - DataIntersectionOf( xsd:nonNegativeInteger xsd:nonPositiveInteger )
  - Union of
    - DataUnionOf( xsd:string xsd:integer )
  - Complement of data range
    - DataComplementOf( xsd:positiveInteger )

An example

:Teenager a
  [owl:Restriction ;
  owl:onProperty :hasAge ;
  owl:someValuesFrom _:y .]
_:y a rdfs:Datatype ;
  owl:onDatatype xsd:integer ;
  owl:withRestrictions ( _:z1 _:z2 ) .
_:z1 xsd:minInclusive "13"^^xsd:integer .
_:z2 xsd:maxInclusive "19"^^xsd:integer .

Punning

- OWL 1 DL things can’t be both a class and an instance
  - E.g., :Snow Leopard can’t be both a subclass of :Feline and an instance of :EndangeredSpecies
- OWL 2 DL offers better support for meta-modeling via punning
  - A URI denoting an owl thing can have two distinct views, e.g., as a class and as an instance
  - The one intended is determined by its use
  - A pun is often defined as a joke that exploits the fact that a word has two different senses or meanings

Punning Restrictions

- Classes and object properties also can have the same name
  - For example, :mother can be both a property and a class of people
- But classes and datatype properties can not have the same name
- Also datatype properties and object properties can not have the same name
Punning Example

@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

foaf:Person a owl:Class.
:Woman a owl:Class.
:Parent a owl:Class.
:mother a owl:ObjectProperty;
   rdfs:domain foaf:Person;
   rdfs:range foaf:Person .
:mother a owl:Class;
   owl:intersectionOf (:Woman :Parent).

validate via http://owl.cs.manchester.ac.uk/validator/

Annotations

• In OWL annotations comprise information that carries no official meaning
• Some properties in OWL 1 are considered as annotation properties, e.g., owl:comment, rdf:label and rdf:seeAlso
• OWL 1 allowed RDF reification as a way to say things about triples, again w/o official meaning
[ a rdf:Statement;
   rdf:subject :Barack_Obama;
   rdf:predicate dbpo:born_in;
   rdf:object :Kenya;
   :certainty "0.01" ].

Annotations

• OWL 2 has native support for annotations, including
  – Annotations on owl axioms (i.e., triples)
  – Annotations on entities (e.g., a Class)
  – Annotations on annotations
• The mechanism is again reification

Annotations

:Man rdfs:subClassOf :Person .
_:x rdf:type owl:Axiom ;
   owl:subject :Man ;
   owl:predicate rdfs:subClassOf ;
   owl:object :Person ;
   :probability "0.99"^^xsd:integer ;
   rdfs:label "Every man is a person." .
Inverse object properties

- Some object property can be inverse of another property
- For example, partOf and hasPart
- The ObjectInverseOf( :partOf ) expression represents the inverse property of :part of
- This makes writing ontologies easier by avoiding the need to name an inverse

OWL Sub-languages

- OWL 1 had sub-languages: OWL FULL, OWL DL and OWL Lite
- OWL FULL is undecidable
- OWL DL is worst case highly intractable
- Even OWL Lite turned out to be not very tractable (EXPTIME-complete)
- OWL 2 introduced three sub-languages, called profiles, designed for different use cases

OWL 2 Profiles

OWL 2 defines three different tractable profiles:
- **EL**: polynomial time reasoning for schema and data
  - Useful for ontologies with large conceptual part
- **QL**: fast (logspace) query answering using RDBMs via SQL
  - Useful for large datasets already stored in RDBs
- **RL**: fast (polynomial) query answering using rule-extended DBs
  - Useful for large datasets stored as RDF triples

OWL Profiles

- Profiles considered
  - Useful computational properties, e.g., reasoning complexity
  - Implementation possibilities, e.g., using RDBs
- There are three profiles
  - OWL 2 EL
  - OWL 2 QL
  - OWL 2 RL
OWL 2 EL

• A (near maximal) fragment of OWL 2 such that
  – Satisfiability checking is in PTime (PTime-Complete)
  – Data complexity of query answering is PTime-Complete
• Based on EL family of description logics
  – Existential (someValuesFrom) + conjunction
• It does not allow disjunction and universal restrictions
• Satisfation is an efficient reasoning technique
• It can capture the expressive power used by many large-scale ontologies, e.g., SNOMED CT

Basic Saturation-based Technique

Normalise ontology axioms to standard form:
\[ A \sqsubseteq B \quad A \sqcap B \sqsubseteq C \quad A \sqsubseteq \exists R.B \quad \exists R.B \sqsubseteq C \]

• Saturate using inference rules:
\[
\begin{align*}
A \sqsubseteq B & \quad B \sqsubseteq C & \quad A \sqsubseteq C \\
A \sqsubseteq B & \quad A \sqsubseteq C & \quad B \sqcap C \sqsubseteq D \\
A \sqsubseteq C & \quad A \sqsubseteq D \\
A \sqsubseteq \exists R.B & \quad B \sqsubseteq C & \quad \exists R.C \sqsubseteq D \\
A \sqsubseteq D \\
\end{align*}
\]

• Extension to Horn fragment requires (many) more rules

Saturation is a general reasoning technique in which you first compute the deductive closure of a given set of rules and add the results to the KB. Then run your prover.

Saturation-based Technique (basics)

Example: infer that a heart transplant is a kind of organ transplant

\[
\text{OrganTransplant} \sqsubseteq \text{Transplant} \sqcap \exists \text{site.Organ} \\
\text{HeartTransplant} \sqsubseteq \text{Transplant} \sqcap \exists \text{site.Heart} \\
\text{Heart} \sqsubseteq \text{Organ}
\]

Saturation-based Technique (basics)

Example:

\[
\text{OrganTransplant} \sqsubseteq \text{Transplant} \sqcap \exists \text{site.Organ} \\
\text{HeartTransplant} \sqsubseteq \text{Transplant} \sqcap \exists \text{site.Heart} \\
\text{Heart} \sqsubseteq \text{Organ}
\]
Saturation-based Technique (basics)

Example:

\[
\begin{align*}
\text{OrganTransplant} & \models \text{Transplant} \land \exists \text{site.Organ} \\
\text{HeartTransplant} & \models \text{Transplant} \land \exists \text{site.Heart} \\
\text{Heart} & \models \text{Organ} \\
\text{OrganTransplant} & \models \text{Transplant} \\
\text{OrganTransplant} & \models \exists \text{site.Organ} \\
\text{Transplant} & \models \exists \text{site.Organ} \land \exists \text{site.Organ} \\
\text{Transplant} & \models \text{OrganTransplant}
\end{align*}
\]
Saturation-based Technique (basics)

Example:

\[
\begin{align*}
\text{OrganTransplant} & \equiv \text{Transplant} \land \exists \text{site.Organ} \\
\text{HeartTransplant} & \equiv \text{Transplant} \land \exists \text{site.Heart} \\
\text{Heart} & \equiv \text{Organ} \\
\exists \text{site.Organ} & \equiv \text{SO} \\
\text{Transplant} \land \text{SO} & \equiv \text{OrganTransplant} \\
\text{HeartTransplant} & \equiv \text{Transplant} \\
\exists \text{site.Heart} & \equiv \text{SH} \\
\text{Transplant} \land \text{SH} & \equiv \text{HeartTransplant} \\
\end{align*}
\]
Saturation-based Technique (basics)

Example:

```
OrganTransplant = Transplant \land \exists \text{site.Organ} \\
HeartTransplant = Transplant \land \exists \text{site.Heart} \\
\text{Heart} \sqsubseteq \text{Organ} \\
\text{OrganTransplant} \sqsubseteq \text{Transplant} \\
\text{OrganTransplant} \sqsubseteq \exists \text{site.Organ} \\
\text{Transplant} \sqcap SO \sqsubseteq \text{OrganTransplant} \\
\text{HeartTransplant} \sqsubseteq \exists \text{site.Heart} \\
\exists \text{site.Heart} \sqsubseteq \text{SH} \\
\text{Transplant} \sqcap \text{SH} \sqsubseteq \text{HeartTransplant} \\
\text{Heart} \sqsubseteq \text{Organ}
```

Galen and SNOMED are large ontologies of medical terms; both have OWL versions. NCI is a vocabulary of cancer-related terms. GO is the gene ontology.

Saturation-based Technique

Performance with large bio-medical ontologies

<table>
<thead>
<tr>
<th>GO</th>
<th>NCI</th>
<th>Galen v.0</th>
<th>Galen v.7</th>
<th>SNOMED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts:</td>
<td></td>
<td>20465</td>
<td>27652</td>
<td>2748</td>
</tr>
<tr>
<td>FACT++</td>
<td>15.24</td>
<td>6.05</td>
<td>465.35</td>
<td>—</td>
</tr>
<tr>
<td>HERMIT</td>
<td>199.52</td>
<td>169.47</td>
<td>45.72</td>
<td>—</td>
</tr>
<tr>
<td>PELLET</td>
<td>72.02</td>
<td>26.47</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CEL</td>
<td>1.84</td>
<td>5.76</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CB</td>
<td>1.17</td>
<td>3.57</td>
<td>0.32</td>
<td>9.58</td>
</tr>
<tr>
<td>Speed-Up:</td>
<td>1.57X</td>
<td>1.61X</td>
<td>143X</td>
<td>∞</td>
</tr>
</tbody>
</table>

**OWL 2 QL**

- The QL acronym reflects its relation to the standard relational Query Language
- It does not allow *existential* and *universal restrictions* to a class expression or a data range
- These restrictions
  - enable a tight integration with RDBMSs,
  - reasoners can be implemented on top of standard relational databases
- Can answer complex queries (in particular, unions of conjunctive queries) over the instance level (ABox) of the DL knowledge base
**OWL 2 QL**
We can exploit **query rewriting** based reasoning technique

— Computationally optimal
— Data storage and query evaluation can be delegated to standard RDBMS
— Can be extended to more expressive languages (beyond AC°) by delegating query answering to a Datalog engine

**Query Rewriting Technique (basics)**

• Given ontology O and query Q, use O to rewrite Q as Q° such that, for any set of ground facts A:
  \[ \text{ans}(Q, O, A) = \text{ans}(Q°, ;, A) \]

• Resolution based query rewriting
  — **Clausify** ontology axioms
  — **Saturate** (clausified) ontology and query using resolution
  — **Prune** redundant query clauses

**Query Rewriting Technique (basics)**

• Example:

\[
\text{Doctor} \sqsubseteq \text{treats}, \text{Patient} \\
\text{Consultant} \sqsubseteq \text{Doctor} \\

Q(x) \leftarrow \text{treats}(x, y) \land \text{Patient}(y)
\]

\(Q(x)\) is our query: Who treats people who are patients?

**Query Rewriting Technique (basics)**

• Example:

\[
\text{Doctor} \sqsubseteq \text{treats}, \text{Patient} \\
\text{Consultant} \sqsubseteq \text{Doctor} \\
\text{treats}(x, f(x)) \leftarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) \leftarrow \text{Doctor}(x) \\
\text{Doctor}(x) \leftarrow \text{Consultant}(x) \\

Q(x) \leftarrow \text{treats}(x, y) \land \text{Patient}(y)
\]

Translate the DL expressions into rules.
Note the use of \(f(x)\) as a Skolem individual. If you are a doctor then you treat someone and that someone is a patient
### Query Rewriting Technique (basics)

**Example:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor (\sqsubseteq) treats.Patient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consultant (\sqsubseteq) Doctor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{treats}(x,f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Patient}(f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor(x) (\rightarrow) Consultant(x)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each rule in the rules version of the KB we want to enhance the query, so that we need not use the rule in the KB.

### Query Rewriting Technique (basics)

**Example:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor (\sqsubseteq) treats.Patient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consultant (\sqsubseteq) Doctor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{treats}(x,f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Patient}(f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor(x) (\rightarrow) Consultant(x)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since Doctor(x) implies \(\text{treats}(x,f(x))\) we can replace it, but we have to also unify \(f(x)\) with \(y\), so we end up with the second way of satisfying our query \(Q(x)\).

### Query Rewriting Technique (basics)

**Example:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor (\sqsubseteq) treats.Patient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consultant (\sqsubseteq) Doctor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{treats}(x,f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Patient}(f(x))) (\rightarrow) Doctor(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor(x) (\rightarrow) Consultant(x)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Applying the KB second rule to the 1st query rule gives us another way to solve the \(Q(x)\).
Query Rewriting Technique (basics)

- Example:

\[
\begin{align*}
\text{Doctor} & \sqsubseteq \text{treats.Patient} \\
\text{Consultant} & \sqsubseteq \text{Doctor}
\end{align*}
\]

\[
\begin{align*}
\text{treats}(x, f(x)) & \leftarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) & \leftarrow \text{Doctor}(x) \\
\text{Doctor}(x) & \leftarrow \text{Consultant}(x)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \leftarrow \text{treats}(x, y) \land \text{Patient}(y) \\
Q(x) & \leftarrow \text{Doctor}(x) \land \text{Patient}(f(x)) \\
Q(x) & \leftarrow \text{treats}(x, f(x)) \land \text{Doctor}(x)
\end{align*}
\]

Since \(\text{Doctor}(x)\) implies \(\text{treats}(x, f(x))\) we can derive \(Q(X)\) if \(\text{Doctor}(x)\) and \(\text{Doctor}(x)\), which reduces to the third query rule.

Query Rewriting Technique (basics)

- Example:

\[
\begin{align*}
\text{Doctor} & \sqsubseteq \text{treats.Patient} \\
\text{Consultant} & \sqsubseteq \text{Doctor}
\end{align*}
\]

\[
\begin{align*}
\text{treats}(x, f(x)) & \leftarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) & \leftarrow \text{Doctor}(x) \\
\text{Doctor}(x) & \leftarrow \text{Consultant}(x)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \leftarrow \text{treats}(x, y) \land \text{Patient}(y) \\
Q(x) & \leftarrow \text{Doctor}(x) \land \text{Patient}(f(x)) \\
Q(x) & \leftarrow \text{treats}(x, f(x)) \land \text{Doctor}(x)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \leftarrow \text{Doctor}(x) \\
Q(x) & \leftarrow \text{Doctor}(x) \land \text{Patient}(f(x)) \\
Q(x) & \leftarrow \text{treats}(x, f(x)) \land \text{Doctor}(x)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \leftarrow \text{Doctor}(x) \\
Q(x) & \leftarrow \text{Consultant}(x)
\end{align*}
\]
Query Rewriting Technique (basics)

• Example:

\[
\begin{align*}
\text{Doctor} & \equiv \text{treats. Patient} \\
\text{Consultant} & \equiv \text{Doctor} \\
\text{treats}(x, f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Doctor}(z) & \rightarrow \text{Consultant}(z)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \equiv \text{treats}(x, y) \land \text{Patient}(y) \\
Q(x) & \equiv \text{Doctor}(x) \land \text{Patient}(f(x)) \\
Q(x) & \equiv \text{treats}(x, f(z)) \land \text{Doctor}(z) \\
Q(z) & \equiv \text{Doctor}(z) \\
Q(z) & \equiv \text{Consultant}(z)
\end{align*}
\]

Remove useless redundant query rules

Query Rewriting Technique (basics)

• Example:

\[
\begin{align*}
\text{Doctor} & \equiv \text{treats. Patient} \\
\text{Consultant} & \equiv \text{Doctor} \\
\text{treats}(x, f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Doctor}(z) & \rightarrow \text{Consultant}(z)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \equiv \text{treats}(x, y) \land \text{Patient}(y) \\
Q(x) & \equiv \text{Doctor}(x) \land \text{Patient}(f(x)) \\
Q(x) & \equiv \text{treats}(x, f(z)) \land \text{Doctor}(z) \\
Q(z) & \equiv \text{Doctor}(z) \\
Q(z) & \equiv \text{Consultant}(z)
\end{align*}
\]

Data can be stored/left in RDBMS
• Relationship between ontology and DB defined by mappings, e.g.:

\[
\begin{align*}
\text{Doctor} & \rightarrow \text{SELECT Name FROM Doctor} \\
\text{Patient} & \rightarrow \text{SELECT Name FROM Patient} \\
\text{treats} & \rightarrow \text{SELECT DName, PName FROM Treats}
\end{align*}
\]

• UCQ translated into SQL query:

\[
\begin{align*}
\text{SELECT Name FROM Doctor UNION SELECT DName FROM Treats, Patient WHERE PName=Name}
\end{align*}
\]

Query Rewriting Technique (basics)

• Example:

\[
\begin{align*}
\text{Doctor} & \equiv \text{treats. Patient} \\
\text{Consultant} & \equiv \text{Doctor} \\
\text{treats}(x, f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Patient}(f(x)) & \rightarrow \text{Doctor}(x) \\
\text{Doctor}(z) & \rightarrow \text{Consultant}(z)
\end{align*}
\]

\[
\begin{align*}
Q(x) & \equiv \text{treats}(x, y) \land \text{Patient}(y) \\
Q(z) & \equiv \text{Doctor}(z) \land \text{Patient}(f(x)) \\
Q(x) & \equiv \text{treats}(x, f(z)) \land \text{Doctor}(z) \\
Q(z) & \equiv \text{Doctor}(z) \\
Q(z) & \equiv \text{Consultant}(z)
\end{align*}
\]

For DL-Lite, result is a union of conjunctive queries (UCQ)
OWL 2 RL

- The RL acronym reflects its relation to Rule Languages
- OWL 2 RL is designed to accommodate
  - OWL 2 applications that can trade the full expressivity of the language for efficiency
  - RDF(S) applications that need some added expressivity from OWL 2
- Not allowed: existential quantification to a class, union and disjoint union to class expressions
- These restrictions allow OWL 2 RL to be implemented using rule-based technologies such as rule extended DBMSs, Jess, Prolog, etc.

Profiles

Profile selection depends on
- Expressiveness required by the application
- Priority given to reasoning on classes or data
- Size of the datasets

Key OWL 2 Documents

http://w3.org/TR/2009/WD-owl2-overview-20090421/
Conclusion

• Most of the new features of OWL 2 in comparing with the initial version of OWL have been discussed
• Rationale behind the inclusion of the new features have also been discussed
• Three profiles – EL, QL and RL – are provided that fit different use cases and implementation strategies