INF3580 – Semantic Technologies – Spring 2010

Lecture 7: The Jena Inference system. OWL introduction

Audun Stolpe

9th February 2010

Department of Informatics

University of Oslo

Today's Plan

Outline

The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
- Three different types of reasoners:
  - Built-in reasoners,
  - External reasoners (Pellet, Fact++, a. o.)
  - DIG reasoners,
    - XML standard for access to description logic processing via HTTP.
    - (not covered here)

Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
  - I. Import and use a known factory class,
    - works for built-in and external reasoners alike
  - II. use a convenience method on the registry
  - III. retrieve a reasoner from the registry using the reasoners URI index
    - suitable for built-in reasoners
- The reasoner can then be applied to a model,
  - to produce an `InfModel`,
  - by applying the reasoner to a plain `Model`,
  - using `ModelFactory.createInfModel(reasoner, model)`

Contd.

One can also construct an `InfModel` in one go

- by using convenience methods on the `ModelFactory` class
  - e.g. `ModelFactory.createRDFSModel(model)`.
- This is typically very simple,
- but makes it more difficult to configure the reasoner

`ModelFactory` also has convience methods that return an `OntModel`

- the `OntModel` class is a subclass of `InfModel`
- has a richer API,
- and can be configured with an `OntModelSpec` parameter
- by calling `ModelFactory.createOntologyModel(param, model)`.

Outline

Built-in reasoners

Included in the Jena distribution are a number of predefined reasoners:

Transitive reasoner: Provides support for simple taxonomy traversal.

- Implements only the reflexivity and transitivity of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`.

RDFS rule reasoner: Supports most of the axioms and inference rules specific to RDFS.

OWL, OWL mini/micro reasoners: Implementations of different subsets of OWL (Lite).

Generic rule reasoner: A rule-based reasoner that supports user defined rules.

Using convenience methods on ModelFactory

Creating a simple RDFSModel

```
Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

- `createRDFSModel()` returns an `InfModel`.
- An `InfModel` supports access to basic inference capability, such as;
    - `getDeductionsModel()` which returns the inferred triples,
    - `getRawModel()` which returns the base triples,
    - `getReasoner()` which returns the RDFS reasoner,
    - `getDerivation(stmt)` which returns the derivation of `stmt`.

Building an InfModel in two steps

The convenience methods on the previous slide builds an `InfModel` in one go.

- We may also build it in the following manner:
    I. Obtain a reasoner first,
    II. Construct a `Model` object (that is, an RDF graph)
    III. pass the reasoner and the model (possibly more than one) to `ModelFactory.createInfModel`

Reasoners are returned by static convenience methods on the registry:

- `ReasonerRegistry.getOWLMicroReasoner()`,
- `ReasonerRegistry.getOWLMiniReasoner()`,
- `ReasonerRegistry.getOWLReasoner()`,
- `ReasonerRegistry.getRDFSReasoner()`,
- `ReasonerRegistry.getRDFSSimpleReasoner()`,
- `ReasonerRegistry.getTransitiveReasoner()`

contd.

using ModelFactory.createInfModel

```
Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

This abstract two-step procedure will be the default, since;

- we retain a reference to the reasoner,
- that can be used for configuration.
- And since it is suitable for built-in and external reasoners alike

Accessing all built-in reasoners

- There are other built-in reasoners than those that are accessible through
    - the convenience methods on `ModelFactory`,
    - and on `ReasonerRegistry`,
    - for instance the `GenericRuleReasoner`.
- All reasoners can be looked up in the registry.
- The `ReasonerRegistry` stores factory instances indexed by URIs.
- Reasoners can be retrieved using these indexes,
- by `registry.create(reasonerURI, param)`
    - where `param` is a configuration parameter,
    - of type `Resource`,
    - but it doesn't do much,
    - and is usually replaced with `null`.

Inspecting the registry

Obtaining an inventory

Get the single global instance of the registry:

```
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
```

Return a description of all reasoners in the form of an RDF graph:

```
Model m = reg.getAllDescriptions();
```

Querying the inventory

```
PREFIX jr:  <http://jena.hpl.hp.com/2003/JenaReasoner#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?reasoner ?desc WHERE {
    ?reasoner rdf:type jr:ReasonerClass .
    ?reasoner jr:description ?desc .
}
```


InfModels by lookup

Reasoners and descriptions

reasoner	desc
jr:DIGReasoner	"Adapter for external (i.e. non-Jena) DIG reasoner"
jr:GenericRuleReasoner	"Generic rule reasoner, configurable"
jr:OWLFBRuleReasoner	"Experimental OWL reasoner. Can separate tbox ..."
jr:OWLMiniFBRuleReasoner	"Experimental mini OWL reasoner. Can separate tbox .."
jr:OWLMicroFBRuleReasoner	"Experimental mini OWL reasoner. Can separate .."
jr:TransitiveReasoner	"Provides reflexive-transitive closure of subClassOf ..."
jr:RDFSExptRuleReasoner	"Complete RDFS implementation supporting ..."
jr:DAMLMicroReasonerFactory	"RDFS rule set with small extensions to support DAML"

Retrieveing a reasoner by URI

```
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
Reasoner r = reg.create("jr:OWLFBRuleReasoner", null);
InfModel inf = ModelFactory.createInfModel(r, sche, dat);
```


Richer models with *OntModel*

- `InfModels` do not enhace the `Model` API as such,
- they only provide basic functionality associated with the reasoner.

An `OntModel` on the other hand

- Provides a better view of a `Model` known to contain ontology data.
- It supplies methods such as
  - `createCardinalityRestriction`,
  - `createSymmetricProperty`,
  - `createRestriction`
- Correspond to language constructs in OWL.
- Required for manipulation of ontologies.

contd.

An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to `ModelFactory`,
- only this time we do not supply a reasoner as an argument,
- rather we supply a model specification,
- which is an `OntModelSpec` object,
- that encapsulates a description of `OntModel` components;
  - the storage scheme,
  - language profile,
  - and the reasoner
- It is thus quite flexible and extensible.

Some specs from OntModelSpec

The class `OntModelSpec` contains static descriptive fields:

`OWL_DL_MEM_RDFS_INF`: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.

`OWL_LITE_MEM`: A specification for OWL Lite models that are stored in memory and do no entailment additional reasoning.

`OWL_MEM_MICRO_RULE_INF`: A specification for OWL models that are stored in memory and use the micro OWL rules inference engine for additional entailments

`OWL_DL_MEM`: A specification for OWL DL models that are stored in memory and do no additional entailment reasoning

Creating OntModels with ModelFactory

Specifying an OntModel

```
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

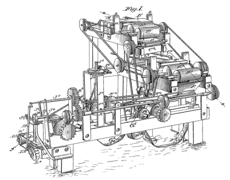
- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
  - then Jena can reason with it (that is, with OWL 2 ontologies),
  - but there may not be support in the API for all language constructs,
  - parts of the ontology may not be *directly* accessible from the code.
  - Likely to change with new releases of Jena.

Outline

Using an external reasoner

External reasoners are are best manipulated directly, that is

- One goes directly to the `FactoryClass`,
- calls the static `theInstance()` to get the factory instance,
- calls the instance's `create()` method,
- and gets the associated reasoner in return.



External reasoners can be combined with `InfModels` and `OntModels` alike.

contd.

In the former case, things are very simple:

Using Pellet with an InfModel

```
Reasoner reas = PelletReasonerFactory.theInstance().create();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

The latter case requires a little more tweaking:

Using Pellet with an OntModel

```
Reasoner r = PelletReasonerFactory.theInstance().create();
InfModel mod = ModelFactory.createInfModel(r, s, d);
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel ont = ModelFactory.createOntologyModel(spec, mod);
```


Outline

Configuration in general

Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
- or to turn transitivity off for properties such as `subClassOf`,
- or to log derivations.

In every case you will need a reference to the reasoner, whence

- it is no longer convenient to use the convenience methods in `ModelFactory`.

Specializing the reasoner

The simplest way to configure a reasoner is to specialize it:

- that is, to bind it to a particular ontology.

This is suitable for situations where,

- you want to apply the same schema to several data sets,
- without redoing too many intermediate deductions

Binding Pellet to schema

```
Reasoner r = PelletReasonerFactory.theInstance().create();
Reasoner custom = r.bindSchema(schema);
InfModel inf = ModelFactory.createInfModel(custom, data);
```

A very simple taxonomy

Consider again the RDFS ontology given by:

```
ex:KillerWhale a rdfs:Class .
ex:Mammal a rdfs:Class .
ex:Vertebrate a rdfs:Class .

ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Mammal rdfs:subClassOf ex:Vertebrate .
```

And suppose we assert:

```
ex:Keiko a ex:KillerWhale .
```

Tracing the derivations could be useful for

- debugging,
- automatic explanation.

Logging derivations

Telling the reasoner to log derivations

```
Reasoner r = ReasonerRegistry.getRDFSReasoner();
r.setDerivationLogging(true);
```

Printing derivations

```
PrintWriter out = new PrintWriter(System.out);
StmtIterator it = inf.listStatements();

while(it.hasNext()){
  Statement stat = (Statement) it.next();
  for(Iterator id = inf.getDerivation(stat);id.hasNext();){
    Derivation deriv = (Derivation) id.next();
    deriv.printTrace(out, true);
  }
}
```


A sample trace

```
Rule rdfs9-alt concluded (ex:Keiko rdf:type ex:Vertebrate) <-
   Fact (ex:KillerWhale rdfs:subClassOf ex:Vertebrate)
   Rule rdfs9-alt concluded (ex:Keiko rdf:type ex:KillerWhale) <-
       Fact (ex:KillerWhale rdfs:subClassOf ex:KillerWhale)
       Known (ex:Keiko rdf:type ex:KillerWhale) - already shown
```


Outline

Quick facts

OWL:

- Acronym for *The Web Ontology Language*.
- Became a W3C reccomendation in 2004.
- Enables boolean reasoning over classes and relationships.
- Superseded by OWL 2;
  - a backwards compatible extension that adds new capabilities.
- The OWL family of languages are based on Description Logics.
- DLs have well-understood and attractive computational properties.

Glimpse ahead: OWL profiles

- OWL has various profiles that correspond to different DLs.
- These profiles are tailored for specific ends, e.g.
  - OWL 2 QL:
    - Specifically designed for efficient database integration.
  - OWL 2 EL:
    - A lightweight language with polynomial time reasoning.
    - Much used in mediacl informatics (e.g. the GALEN ontology).
  - OWL 2 RL:
    - Designed for compatibility with rule-based inference tools.

The $\mathcal{ALEC}$ fragment of OWL

$\mathcal{ALEC}$ In DL-notation

$$
\begin{array}{rlll}
C, D \rightarrow & A & \mid & \text{(atomic concept)} \\
& \top & \mid & \text{(universal concept)} \\
& \bot & \mid & \text{(bottom concept)} \\
& \neg C & \mid & \text{(atomic negation)} \\
& C \sqcap D & \mid & \text{(intersection)} \\
& \forall R.C & \mid & \text{(value restriction)} \\
& \exists R.C & \mid & \text{(existential restriction)}
\end{array}
$$

Semantics

$\mathcal{ALEC}$ in DL-notation

$$
\begin{array}{rcl}
\top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} & = & \emptyset \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \forall b(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \exists b(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}
\end{array}
$$

OWL ontologies in DL-notation

$$
\begin{array}{rcl}
Cystic\_Fibrosis & \equiv & Fibrosis \sqcap \exists locatedIn.Pancreas \\
Genetic\_Fibrosis & \sqsubseteq & Genetic\_Disorder \\
Fibrosis \sqcap \exists locatedIn.Pancreas & \sqsubseteq & Genetic\_Fibrosis
\end{array}
$$

Some differences from RDFS

1. Complex classes can be expressed:
   - $C \sqcap D$ corresponds to logical conjunction,
   - $C \sqcup D$ to logical disjunction, and
   - $\neg C$ to logical negation
2. Unlike RDFS, OWL is therefore a boolean language.
   - That is, it has a propositional logic as a fragment.
3. Full propositional negation facilitates consistency checking.

Existential restrictions

- Allow us to describe classes in terms of each other.

  $Cystic\_Fibrosis \equiv Fibrosis \sqcap \exists locatedIn.Pancreas$
- or, more mundanely

  $ProudMother \equiv Woman \sqcap \exists hasChild.Lawyer$
- `hasChild.Lawyer` = the set of things that have at least one lawyer child.
  - If a thing has a lawyer child,
  - and that thing is a woman,
  - then that thing is a proud mother

Existential restrictions in Turtle syntax

Lawyer children

```
[a  owl:Restriction;
    owl:onProperty :hasChild:
    owl:somValuesFrom :Lawyer] .
```

- `owl:Restriction` signals a class description,
- `owl:somValuesFrom`; an existential restriction on a property,
- `owl:onProperty` gives the property
- The description is a blank node, since it has no name.

Existential restrictions illustrated

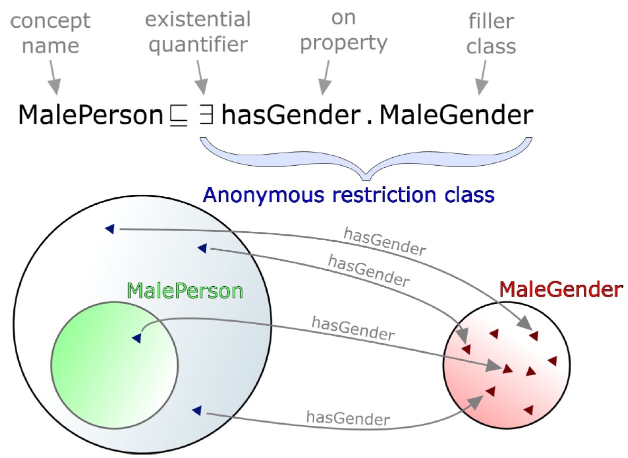


Figure: Existential restrictions

Horisontal relations between classes

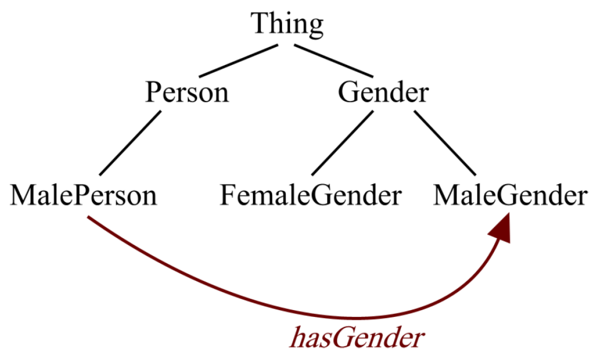


Figure: Existential restrictions relate classes

Returning to an example

Suppose we assert:

1. `:OsloPhilharmonic :conductor :Saraste .`

And we say that

2. Orchestra $\equiv \exists$conductor.$\top \sqcap \exists$hasInstrument.$\top$

Then from [1.] we may infer that

3. `:OsloPhilharmonic a :Orchestra .`
4. `:OsloPhilharmonic :hasInstrument _:x .`

A comparison with `rdfs:domain`

- Recall that `ex:conductor rdfs:domain ex:Orchestra` says that only orchestras have conductors.
- We can express this with existential restrictions:
  $\exists$conductor.$\top \equiv$ Orchestra
- But we can also express a number finer relationships:
  Choir $\sqsubseteq \exists$conductor.$\top$
  $\exists$conductor.Cantor $\sqsubseteq$ ChurchEnsemble
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
- If we add that
  `:MusicaAntiqua :conductor :Savall .` (not actually the case)
  `:Savall a :Cantor` (nor is this)
- then we know that
  - `:MusicaAntiqua a :ChurchEnsemble .` (nope)

Existential restrictions in `OntModels`

Implementing the example

```
OntModel m = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM);
OntClass c = m.createClass("ex:Cantor");
OntClass e = m.createClass("ex:ChurchEnsemble");
ObjectProperty cond = m.createObjectProperty("ex:conductor");
// null denotes the URI in an anonymous restriction
SomeValuesFromRestriction r = m.createSomeValuesFromRestriction(null, cond, c);
Statement stmt = model.createStatement(r,OWL.subClassOf, e);
model.add(stmt);
```

More about this later

Supplementary reading

- The Jena ontology API:
- Jena Inference Engine user manual:
- Using a DIG Description Logic reasoner with Jena:

All available from the Jena website.