# INF3580/4580 – Semantic Technologies – Spring 2017

## Lecture 7: Reasoners in Jena

Martin Giese

27th February 2018

Department of Informatics

University of Oslo

## Today's Plan







## Outline







## What is inference?

In a Semantic Web context, inference always means,

- adding triples,

More specifically it means,

- adding new triples to an RDF graph,
- on the basis of the triples already in it.
- 'adding' should be understood in a logical sense, indeed;
  - new/inferred triples need not be materialized or persisted

## cont.

A rule of the form

$$\frac{P_1, \cdots, P_n}{P}$$

may be read as an instruction;

- "If $P_1, \cdots, P_n$ are all in the graph, add $P$ to the graph"
- as an *instruction* this may in turn be understood *procedurally*...
  - in a forward sense, or
  - in a backward sense





## RDFS reasoning

RDFS supports three principal kinds of reasoning pattern:

I. Type propagation:
   - "The 2CV is a car, and a car is a motorised vehicle, so..."

II. Property inheritance:
   - "Martin lectures at Ifi, and lecturers are employed by Ifi, so..."

III. Domain and range reasoning:
   - "Everything written is a document. Martin wrote $x$, hence $x$..."
   - "All fathers are males. Martin is the father of Karl, therefore..."





## Sample RDFS rules

**Rules for property transfer**

- Transitivity:

$$\frac{\texttt{p rdfs:subPropertyOf q .} \quad \texttt{q rdfs:subPropertyOf r .}}{\texttt{p rdfs:subPropertyOf r .}} \text{rdfs5}$$

- Reflexivity:

$$\frac{\texttt{p rdf:type rdf:Property .}}{\texttt{p rdfs:subPropertyOf p .}} \text{rdfs6}$$

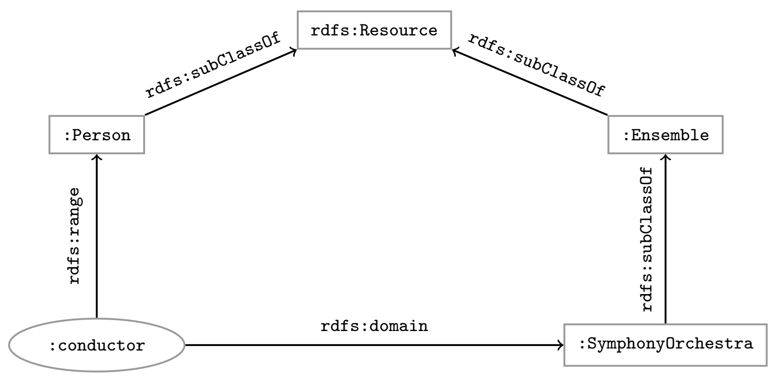
- Property transfer:

$$\frac{\texttt{p rdfs:subPropertyOf q .} \quad \texttt{u p v .}}{\texttt{u q v .}} \text{rdfs7}$$





## Example: Conductors and ensembles



## Example contd.

This ontolology includes

```
:SymphonyOrchestra rdfs:subClassOf :Ensemble .
:conductor rdfs:domain :SymphonyOrchestra .
:conductor rdfs:range :Person .
```

Suppose the data includes

```
:OsloPhilharmonic :conductor :Petrenko .
```

then the the following triples can be inferred:

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
:OsloPhilharmonic rdf:type :Ensemble .
:Petrenko rdf:type :Person .
```

try to figure out why!





## Outline







## Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Backward chaining:

- reasoning from conclusions to premises
- '. . . what needs to be true for this conclusion to hold?'
- reasoning is on-demand





## Forward chaining inference

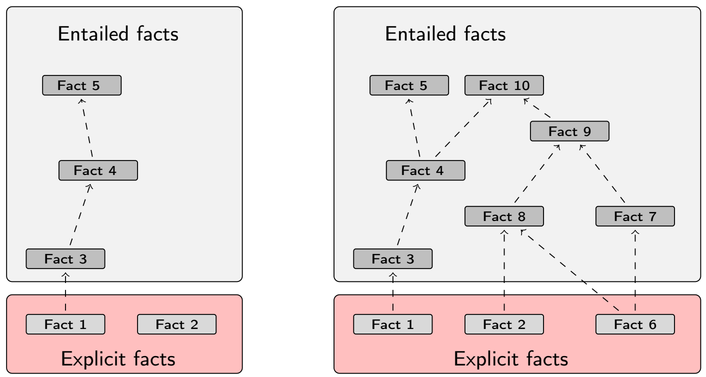



Figure: When a fact is added, all entailments are computed and stored.

## Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,
- and small enough to store efficiently.

Benefits:

- forward chaining optimizes retrieval
- no additional inference is necessary at query time





## Forward chaining and truth-maintenance

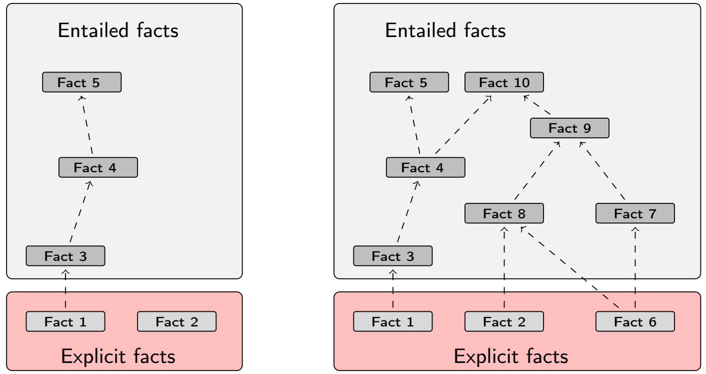



Figure: When a fact is added, all entailments are computed and stored.





## Forward chaining and truth-maintenance

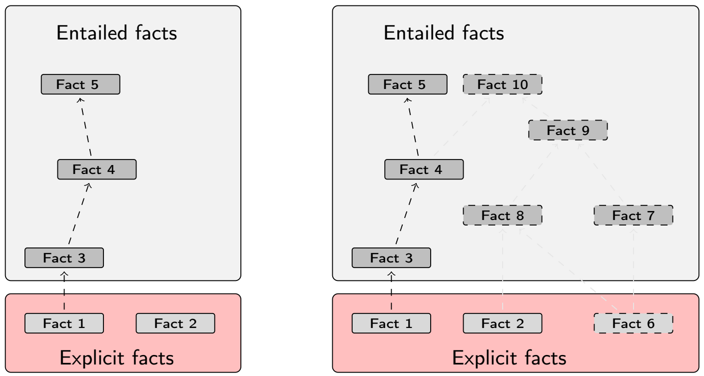



Figure: When a fact is removed, everything that comes with it must go too.





## Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic
- truth maintenance usually not implemented in RDF stores
- problematic for distributed and/or dynamic systems
  - rules could apply to premisses on different disks, etc.

## Backward chaining inference

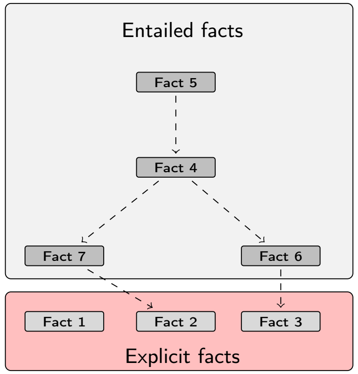



Figure: Backward chaining uses rules to expand queries.





## Backward chaining: Example

**RDFS/RDF knowledge base:**

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

ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

$$\frac{\texttt{A rdfs:subClassOf B .} \quad \texttt{x rdf:type A .}}{\texttt{x rdf:type B .}}$$

**Query:**

```
SELECT ?x WHERE { ?x rdf:type ex:Vertebrate . }
```

**Inferred triples:**

`?x rdf:type ex:Vertabrate .`

`?x rdf:type ex:Mammal .` (rdfs9)

`?x rdf:type ex:KillerWhale .` (rdfs9) $\Rightarrow$ `?x = ex:Keiko`

`?x rdf:type ex:Lion .` (rdfs9) $\Rightarrow$ `?x = ex:Simba`





## Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic
- no persistent storage space needed

Drawbacks:

- trades insertion overhead for access overhead
- without caching, answers must be recomputed every time





## Outline

## Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`
- Different reasoners implement different logics, e.g
  - Transitive reasoning,
  - RDFS,
  - OWL





## Reasoners, Factories, Registries...

- Every reasoner is an object of class `Reasoner`
- These are created by `ReasonerFactory` objects
- So: one `ReasonerFactory` per type of reasoner
- All reasoner factories are stored in a global `ReasonerRegistry`
  - Allows finding a factory for reasoners by URI
  - Also by "descriptions" which are again RDF
- Example:

```
ReasonerRegistry registry = ReasonerRegistry.theRegistry();
String reasonerURI = "http://jena.hpl.hp.com/2003/RDFSExptRuleReasoner";
ReasonerFactory factory = registry.getFactory(reasonerURI);
Reasoner reasoner = factory.create(config);
```

- `config` is a `Resource` that describes requested features for the reasoner.





## Inference Models

- Now a `Model` with inferencing can be constructed, given
  - an underlying `Model` with "raw" data
  - a `Reasoner` instance

```
InfModel inf = ModelFactory.createInfModel(reasoner, rawModel);
```

- Depending on reasoner, this `InfModel` might do
  - forward chaining: precompute all consequences of triples in `rawModel`
  - backward chaining: triggered by SPARQL queries or `list(...)` calls
- Different reasoners compute different sets of consequences:
  - "transitive" reasoner: only `subClassOf` hierarchy, etc.
  - RDFS reasoner: all RDFS inference rules
  - OWL/mini/micro: various subsets of OWL inferences
- Most reasoners can be configured before binding them to a model, to change various details of their behaviour.





## The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
```

- Get inference models with standard reasoners from `ModelFactory`:

```
InfModel inf = ModelFactory.createRDFSModel(rawModel);
```

- What's the point of the long winded way?
  - Can ask for non-builtin provers, e.g. Pellet
  - Can configure reasoners

## Simplified overview

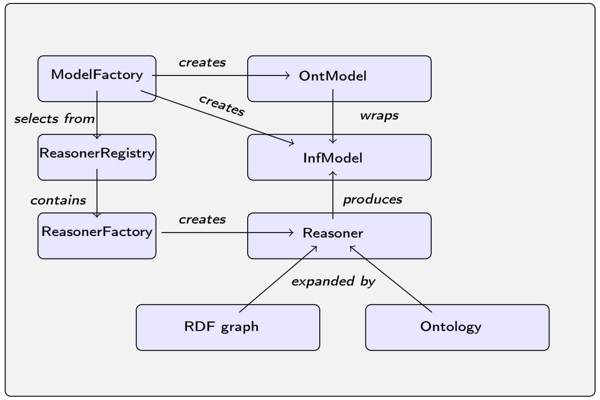



Figure: The structure of the reasoning system





## Outline







## Built-in reasoners

Transitive reasoners:

- provides support for simple taxonomy traversal
- implements only the reflexivity and transitivity of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`.

RDFS reasoners:

- supports (most of) the axioms and inference rules specific to RDFS.

OWL, OWL mini/micro reasoners:

- implements different subsets of the OWL specification





## Obtaining a built-in reasoner

Three main ways of obtaining a built-in reasoner:

1. call a convenience method on the `ModelFactory`
   - which calls a `ReasonerFactory` in the `ReasonerRegistry`, and
   - returns an `InfModel` all in one go
2. call a static method in the `ReasonerRegistry`,
   - the static method returns a reasoner object
   - pass it to `ModelFactory.createInfModel()`
   - along with a model and a dataset
3. use a reasoner factory directly
   - covered in connection with external reasoners later

## Example I: Using a convenience method

### A simple RDFS model

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

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;
  - `getDeductionsModel()` which returns the inferred triples,
  - `getRawModel()` which returns the base triples,
  - `getReasoner()` which returns the RDFS reasoner,
  - `getDerivation(stmt)` which returns a trace of the derivation





## Example II: Using static methods in the registry

### 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);
```

Virtues of this approach:

- we retain a reference to the reasoner,
- that can be used to configure it
  - e.g. to do backwards or forwards chaining
  - . . . mind you, not all reasoners can do both
- similar for built-in and external reasoners alike





## Outline







## An *OntModel* is ontology-aware

An `InfModel` provides

- basic functionality associated with the reasoner, and
- basic functionality to sort entailed from explicit statements
- . . . but no fine-grained control over an ontology

An `OntModel` provides

- a richer view of a knowledge base
- in terms of ontological concepts
- mirrored by methods such as
  - `createClass()`
  - `createDatatypeProperty()`
  - `getIntersectionClass()`

## contd.

An *OntModel* does not by itself compute entailments

- it is merely a wrapper
- that provides a convenient API
- given that your data is described by an ontology

However,

- an `OntModel` can be constructed according to a specification object
- that, among other things, tells Jena which reasoner to use

More generally, an `OntModelSpec` encapsulates

- the storage scheme,
- language profile,
- and the reasoner associated with a particular `OntModel`





## Some predefined specification objects

The class `OntModelSpec` contains static references to prebuilt instances:

`OWL_DL_MEM_RDFS_INF`: In-memory OWL DL model that uses the RDFS inference engine.
`OWL_LITE_MEM`: In-memory OWL Lite model. No reasoning.
`OWL_MEM_MICRO_RULE_INF`: In-memory OWL model uses the OWLMicro inference engine.
`OWL_DL_MEM`: In-Memory OWL DL model. No reasoning.





## Example: Configuring an OntModel

An `OntModel` is created by calling a method in `ModelFactory`

**Specifying an OntModel**

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

Jena currently lags behind (…and has done so for quite a while)

- no spec for OWL 2
- …or any of its profiles
- does not mean that we cannot use OWL 2 ontologies with Jena
- but we do not have support in the API for all language constructs
- some reasoners supply their own such API, e.g. Pellet





## Question

So… we learnt how to use Jena to add, retrieve, modify triples
— why do we need reasoners?

Many reasons:

- Separate logic (All symphony orchestras are ensembles) from control (when to add which triples): declarative programming.
- Can use ontology reasoners to check that the logic is OK. Much easier than checking that a Java program is OK.
- Getting the control right (and efficient) is not always easy. Using a generic reasoner reuses this know-how.

## Outline







## Plugging in third-party reasoners

Jena's reasoning-system architecture makes it easy...

- for third party vendors to write reasoners
- that can be plugged in to Jena architecture

External reasoners usually

- check in a `ReasonerFactory` in the `ReasonerRegistry`, and
- supply a `OntModelSpec` to be handed to the `ModelFactory`





## Some better known ones

There are many, many reasoners to choose from, e.g.

- FaCT++
- Cerebra Engine
- CEL
- HermiT
- Pellet

Reasoning algorithms vary with purpose, scope, philosophy and age (!);

- tableau reasoners (FaCT++, Pellet, Cerebra)
- rule-based reasoners (CEL)
- hyper-tableaux (HermiT)
- only rule reasoners have a notion of forwards vs. backwards





## Using an external reasoner

- retrieve an instance of the reasoner:

```
Reasoner r;
r = PelletReasonerFactory.theInstance().create();
```

- associate the reasoner with an `InfModel`, an ontology and a dataset:

```
InfModel inf;
inf = ModelFactory.createInfModel(r, ontology, dataset);
```

- Or: create an `OntModel` for a richer API:

```
OntModel m;
m = ModelFactory.createOntologyModel(
    PelletReasonerFactory.THE_SPEC);
```

## Outline







## Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content from Wikipedia
- it is a community effort, so...
- the data is not always uniform and consistent
- distinct properties for 'intuitively similar' objects not uncommon, e.g.;
  - `dbprop:doctoralStudents`
  - `dbpedia:doctoralStudent`





## Who has worked with Jeffrey Ullman?

Ullman is one of the most referenced computer scientists

- DBpedia contains info about, e.g. his
  - education and laureates
  - citizenship and nationality
  - scientific contributions
- say we wish to compile a list of his collaborators, including at least
  - advisors, and
  - PhD students





- set relevant prefixes:

```
String ont = "http://dbpedia.org/ontology/";
String res = "http://dbpedia.org/resource/";
String prop = "http://dbpedia.org/property/";
String ex = "http://www.example.org/";
```

- connect to DBpedia, describe J. Ullman:

```
String dbpedia = "http://dbpedia.org/sparql";
String describe = "DESCRIBE <" + res + "Jeffrey_Ullman>";
QueryExecution qexc =
  QueryExecutionFactory.sparqlService(dbpedia, describe);
Model ullman = qexc.execDescribe();
```

- build an ontology of collaborators (or better, read it from file):

```
Model ontology = ModelFactory.createDefaultModel();
Property collab = ontology.createProperty(ex + "collaborator");
Property phds = ontology.createProperty(prop + "doctoralStudents");
Property phd = ontology.createProperty(ont + "doctoralStudent");
Property adv = ontology.createProperty(ont + "doctoralAdvisor");
ontology.add(phds, RDFS.subPropertyOf, collab);
ontology.add(phd, RDFS.subPropertyOf, collab);
ontology.add(adv, RDFS.subPropertyOf, collab);
```

- ...and reason over it:

```
InfModel inf;
inf = ModelFactory.createRDFSModel(ontology, ullman);
```

- wrap it in an `OntModel` if you need a richer API





- write the query:

```
String qStr =
"PREFIX ont:  <" + ont + ">" +
"PREFIX res:  <" + res + ">" +
"PREFIX ex:  <" + ex + ">" +
"SELECT ?collaborator WHERE {" +
" res:Jeffrey_Ullman ex:collaborator ?collaborator." +
"}";
```

- execute it...

```
Query query = QueryFactory.create(qStr);
QueryExecution qe = QueryExecutionFactory.create(query, inf);
ResultSet res = qe.execSelect();
```

- and, if, you like, print out the results

```
ResultSetFormatter.out(res, query);
```





## Backwards reasoning over the same example

- backwards reasoning often suitable for stuff in memory
- you need a reasoner capable of doing backwards reasoning
- i.e. a rule reasoner
- and a way to configure it
- let's use the built-in `RDFSRuleReasoner`
- first create a configuration specification:

```
# A config spec is itself an RDF graph
Resource config = ontology.createResource();
```





- `ReasonerVocabulary` holds terms for configuration purposes:

```
config.addProperty(ReasonerVocabulary.PROPruleMode, "backward");
```

- now create a rule reasoner and pass it the configuration

```
Reasoner r;
r = RDFSRuleReasonerFactory.theInstance().create(config);
```

- proceed as before...

## Next Weeks

- (Simplified) Model Semantics for RDF and RDFS
- Relationship Reasoning $\Longleftrightarrow$ Semantics
- OWL, semantics of that, etc.