INF3580 – Semantic Technologies – Spring 2011

Lecture 7: Reasoners in Jena

Audun Stolpe

8th March 2011

Department of Informatics

University of Oslo

Today's Plan

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. Built-in reasoners
5. Richer API with `OntModel`
6. External reasoners
7. A worked example

Outline

What is inference?

In a Semantic Web context, inference always means,

What is inference?

In a Semantic Web context, inference always means,

- adding triples,

What is inference?

In a Semantic Web context, inference always means,

- adding triples,

More specifically it means,

What is inference?

In a Semantic Web context, inference always means,

- adding triples,

More specifically it means,

- adding new triples to an RDF graph,

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.

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;

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/entailed triples need not be materialized or persisted

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/entailed triples need not be materialized or persisted
  - indeed they may be ephemeral and transitory

cont.

A rule of the form

cont.

A rule of the form

cont.

A rule of the form

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

may be read as an instruction;

cont.

A rule of the form

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

may be read as an instruction;

- "If $P_1, \ldots, P_n$ are all in the graph, add P to the graph"

cont.

A rule of the form

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

may be read as an instruction;

- “If $P_1, \ldots, P_n$ are all in the graph, add P to the graph”
- as an *instruction* this may in turn be understood *procedurally* ...

cont.

A rule of the form

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

may be read as an instruction;

- "If $P_1, \ldots, P_n$ 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

cont.

A rule of the form

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

may be read as an instruction;

- "If $P_1, \ldots, P_n$ 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:

RDFS reasoning

RDFS supports three principal kinds of reasoning pattern:

I. Type propagation:

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…"

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:

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 anyone who does so is employed by Ifi, so..."

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 anyone who does so is employed by Ifi, so..."

III. Domain and range reasoning:

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 anyone who does so is employed by Ifi, so..."

III. Domain and range reasoning:
   - "Everything someone has written is a document. Martin has written a PhD thesis, therefore..."

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 anyone who does so is employed by Ifi, so. . ."

III. Domain and range reasoning:
  - "Everything someone has written is a document. Martin has written a PhD thesis, therefore. . ."
  - "All fathers of people are males. Martin is the father of Karl, therefore. . ."

Sample RDFS rules

Sample RDFS rules

Rules for property transfer

Sample RDFS rules

Rules for property transfer

- Transitivity:

Sample RDFS rules

Rules for property transfer

- Transitivity:

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

Sample RDFS rules

Rules for property transfer

- Transitivity:

$$\frac{\texttt{p rdfs:subPropertyOf q .} \qquad \texttt{q rdfs:subPropertyOf r .}}{\texttt{p rdfs:subPropertyOf r .}}\ \textsf{rdfs5}$$

- Reflexivity:

Sample RDFS rules

Rules for property transfer

- Transitivity:

$$\frac{\texttt{p rdfs:subPropertyOf q .} \qquad \texttt{q rdfs:subPropertyOf r .}}{\texttt{p rdfs:subPropertyOf r .}}\ \textsf{rdfs5}$$

- Reflexivity:

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

Sample RDFS rules

Rules for property transfer

- Transitivity:

$$\frac{\texttt{p rdfs:subPropertyOf q .} \qquad \texttt{q rdfs:subPropertyOf r .}}{\texttt{p rdfs:subPropertyOf r .}}\ \textsf{rdfs5}$$

- Reflexivity:

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

- Property transfer:

Sample RDFS rules

Rules for property transfer

- Transitivity:

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

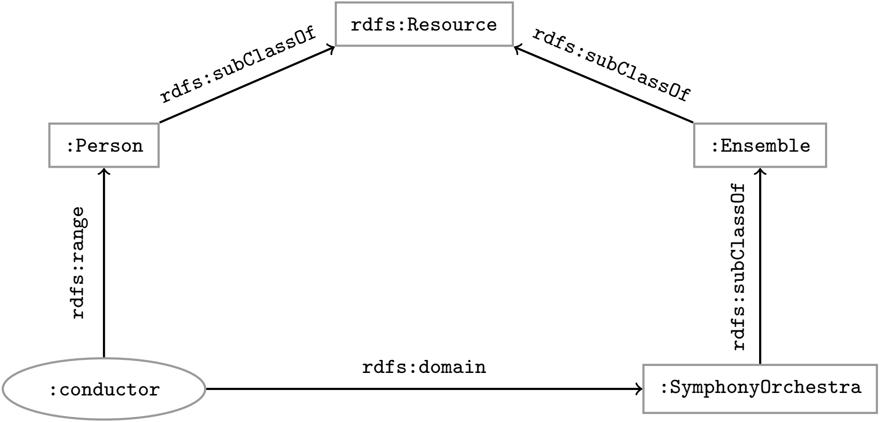
- Reflexivity:

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

- Property transfer:

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

Example: Conductors and ensembles



Example contd.

This ontolology includes

Example contd.

This ontolology includes

```
:SymphonyOrchestra rdfs:subClassOf :Ensemble .
```

Example contd.

This ontolology includes

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

Example contd.

This ontolology includes

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

Example contd.

This ontolology includes

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

the data includes

Example contd.

This ontolology includes

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

the data includes

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

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

the entailments include

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

the entailments include

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

the entailments include

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

Example contd.

This ontolology includes

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

the data includes

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

but interestingly not

```
:OsloPhilharmonic rdf:type :SymphonyOrchestra .
```

the entailments include

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

Outline

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. Built-in reasoners
5. Richer API with `OntModel`
6. External reasoners
7. A worked example

Forward chaining vs. backward chaining

Forward chaining:

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusion of rules

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusion of rules
- adds facts corresponding to the conclusions of rules

Forward chaining vs. backward chaining

Forward chaining:

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

Forward chaining vs. backward chaining

Forward chaining:

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

Backward chaining:

Forward chaining vs. backward chaining

Forward chaining:

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

Backward chaining:

- reasoning from conclusions to premises

Forward chaining vs. backward chaining

Forward chaining:

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

Backward chaining:

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

Forward chaining vs. backward chaining

Forward chaining:

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

Backward chaining:

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

Forward chaining inference

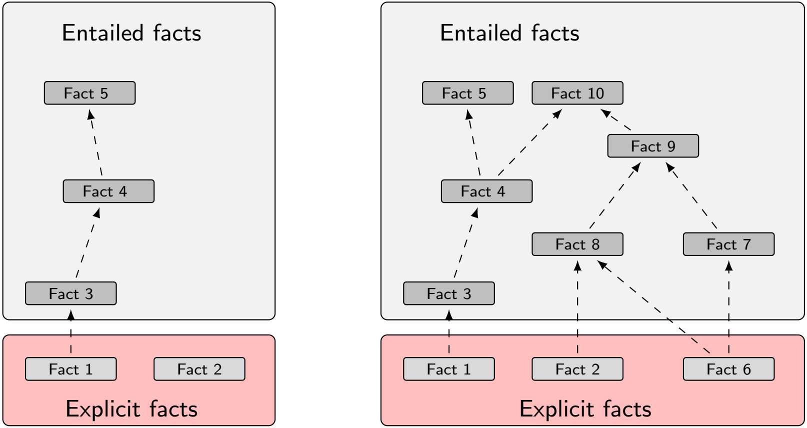


Figure: When a fact is added, all entailments are computed and stored.

Benefits of forward chaining

Precomputing and storing answers is suitable for:

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,
- which is relatively static,

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,
- which is relatively static,
- and which is small enough to efficiently store

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,
- which is relatively static,
- and which is small enough to efficiently store

Benefits:

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,
- which is relatively static,
- and which is small enough to efficiently store

Benefits:

- forward chaining optimizes retrieval

Benefits of forward chaining

Precomputing and storing answers is suitable for:

- frequently accessed data
- which it is expensive to compute,
- which is relatively static,
- and which is small enough to efficiently store

Benefits:

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

Forward chaining and truth-maintenance

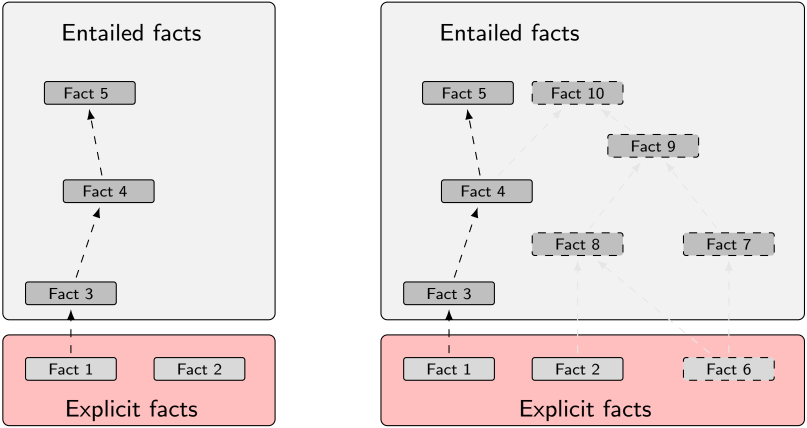


Figure: When a fact is removed, everything that comes with it must go too.

Drawbacks of forward chaining

Drawbacks:

Drawbacks of forward chaining

Drawbacks:

- increases storage size

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic

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

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
- not suitable for distributed and/or dynamic systems

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
- not suitable for distributed and/or dynamic systems
- (... as there is usually nowhere to persist the data)

Backward chaining inference

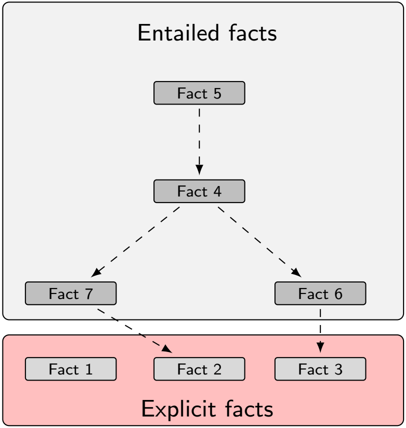


Figure: Backward chaining uses rules to expand queries.

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers

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

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

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:

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

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

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 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:

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

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

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner

Quick facts

In Jena there is

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

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 ...

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

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

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

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

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

Quick facts

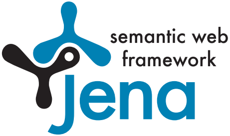
In Jena there is

- 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`

Moreover ...



Moreover ...

- Different reasoners implement different logics, e.g semantic web framework Jena

Moreover ...

- Different reasoners implement different logics, e.g
  - Transitive reasoning,

Moreover ...

- Different reasoners implement different logics, e.g
    - Transitive reasoning,
    - RDFS,

Moreover ...

- Different reasoners implement different logics, e.g
    - Transitive reasoning,
    - RDFS,
    - OWL

Moreover ...

- Different reasoners implement different logics, e.g
    - Transitive reasoning,
    - RDFS,
    - OWL
- There is a `ReasonerFactory` class for each type of reasoner,

Moreover ...

- Different reasoners implement different logics, e.g
  - Transitive reasoning,
  - RDFS,
  - OWL
- There is a `ReasonerFactory` class for each type of reasoner,
  - which is used to create `Reasoner` objects

Moreover ...

- Different reasoners implement different logics, e.g
    - Transitive reasoning,
    - RDFS,
    - OWL
- There is a `ReasonerFactory` class for each type of reasoner,
    - which is used to create `Reasoner` objects
    - they are all stored in a global `ReasonerRegistry` class

Moreover ...

- Different reasoners implement different logics, e.g
    - Transitive reasoning,
    - RDFS,
    - OWL
- There is a `ReasonerFactory` class for each type of reasoner,
    - which is used to create `Reasoner` objects
    - they are all stored in a global `ReasonerRegistry` class
    - which can be manipulated explicitly or implicitly

The road most often travelled ...

Applications normally access the inference machinery by

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner
- producing a new model with reasoning capabilities

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner
- producing a new model with reasoning capabilities

The `ModelFactory` may

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner
- producing a new model with reasoning capabilities

The `ModelFactory` may

- create an `InfModel` via convenience methods on the `Registry`, or

The road most often travelled …

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner
- producing a new model with reasoning capabilities

The `ModelFactory` may

- create an `InfModel` via convenience methods on the `Registry`, or
- create an `OntModel` and pass it an `OntModelSpec`

The road most often travelled ...

Applications normally access the inference machinery by

- using the `ModelFactory`
- to associate a dataset with some reasoner
- producing a new model with reasoning capabilities

The `ModelFactory` may

- create an `InfModel` via convenience methods on the `Registry`, or
- create an `OntModel` and pass it an `OntModelSpec`

.... Confusing? Stay tuned ....

Simplified overview

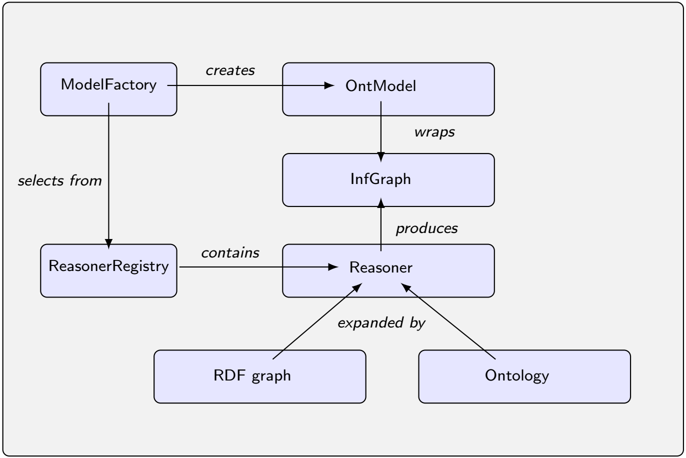


Figure: The structure of the reasoning system

Built-in reasoners

Transitive reasoners:

Built-in reasoners

Transitive reasoners:

- provides support for simple taxonomy traversal

Built-in reasoners

Transitive reasoners:

- provides support for simple taxonomy traversal
- implements only the reflexivity and transitivity of

Built-in reasoners

Transitive reasoners:

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

Built-in reasoners

Transitive reasoners:

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

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:

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.

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:

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

Outline

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. **Built-in reasoners**
5. Richer API with `OntModel`
6. External reasoners
7. A worked example

Obtaining a built-in reasoner

Three main ways of obtaining a built-in reasoner:

Obtaining a built-in reasoner

Three main ways of obtaining a built-in reasoner:

1. call a convenience method on the `ModelFactory`

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

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

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`,

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

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()`

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

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

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);
```

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`

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;

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,

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,

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,

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);
```

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:

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,

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

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

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

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

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. Built-in reasoners
5. **Richer API with `OntModel`**
6. External reasoners
7. A worked example

An *OntModel* is ontology-aware

An `InfModel` provides

An *OntModel* is ontology-aware

An `InfModel` provides

- basic functionality associated with the reasoner, and

An *OntModel* is ontology-aware

An `InfModel` provides

- basic functionality associated with the reasoner, and
- basic functionality to sort entailed from explicit statements

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* 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

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

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

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

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()`

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()`

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

contd.

An *OntModel* does not by itself compute entailments

- it is merely a wrapper

contd.

An *OntModel* does not by itself compute entailments

- it is merely a wrapper
- that provides a convenient API

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

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,

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

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

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

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,

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,

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:

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.

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.

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.

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`

Example: Configuring an `OntModel`

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

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);
```

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

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

- no spec for OWL 2

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

- no spec for OWL 2
- ... or any of its profiles

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

- no spec for OWL 2
- ... or any of its profiles
- does not mean that we cannot use OWL 2 ontologies with Jena

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

- 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

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

- 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

Outline

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. Built-in reasoners
5. Richer API with `OntModel`
6. External reasoners
7. A worked example

Plugging in third-party reasoners

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

Plugging in third-party reasoners

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

- for third party vendors to write reasoners

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

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

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

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.

Some better known ones

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

- FaCT++

Some better known ones

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

- FaCT++
- Cerebra Engine

Some better known ones

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

- FaCT++
- Cerebra Engine
- CEL

Some better known ones

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

- FaCT++
- Cerebra Engine
- CEL
- HermiT

Some better known ones

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

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

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, philsophy and age (!);

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, philsophy and age (!);

- tableu reasoners (FaCt++, Pellet, Cerebra)

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, philsophy and age (!);

- tableu reasoners (FaCt++, Pellet, Cerebra)
- rule-based reasoners (CEL)

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, philsophy and age (!);

- tableu reasoners (FaCt++, Pellet, Cerebra)
- rule-based reasoners (CEL)
- hyper-tableu (HermiT)

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, philsophy and age (!);

- tableu reasoners (FaCt++, Pellet, Cerebra)
- rule-based reasoners (CEL)
- hyper-tableu (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();
```

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);
```

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);
```

- wrap it in an `OntModel` for a richer API:

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

Outline

1. Recap: Reasoning with rules
2. Backwards and forwards reasoning
3. The Jena reasoning system
4. Built-in reasoners
5. Richer API with `OntModel`
6. External reasoners
7. A worked example

Integrating information from DBpedia

Quick facts about the DBpedia project:

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form Wikipedia

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form Wikipedia
- it is a community effort, so ..

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form Wikipedia
- it is a community effort, so ..
- the data is not always uniform and consistent

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form 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.;

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form 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`

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form 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`

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form 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`
- the latter points to individual students represented by URIs

Integrating information from DBpedia

Quick facts about the DBpedia project:

- aims to extract structured content form 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`
- the latter points to individual students represented by URIs
- the former to a *list* of student names in the from of a string

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- DBpedia contains info about, e.g. his

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- DBpedia contains info about, e.g. his
  - education and laureates

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- DBpedia contains info about, e.g. his
    - education and laureates
    - citizenship and nationality

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- DBpedia contains info about, e.g. his
  - education and laureates
  - citizenship and nationality
  - scientific contributions

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- 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

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- 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

Who has worked with Jeffrey Ullman?

Ullman is *the* most referenced computer scientist

- 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/";
```

- 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:

```
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);
```

- 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);
```

- 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);
```

- 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." +
"}";
```

- 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();
```

- 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);
```

An exercise for the reader ...

- substituting Pellet for the RDFS reasoner yields a different result

An exercise for the reader ...

- substituting Pellet for the RDFS reasoner yields a different result
- reason rooted in the respective ‘philosophies’ of RDFS and OWL

An exercise for the reader ...

- substituting Pellet for the RDFS reasoner yields a different result
- reason rooted in the respective ‘philosophies’ of RDFS and OWL
- tracking it down, is an instructive exercise ...

An exercise for the reader ...

- substituting Pellet for the RDFS reasoner yields a different result
- reason rooted in the respective ‘philosophies’ of RDFS and OWL
- tracking it down, is an instructive exercise ...
- which is left for the student

Backwards reasoning over the same example

- bakcwards reasoning often suitable for stuff in memory

Backwards reasoning over the same example

- bakcwards reasoning often suitable for stuff in memory
- you need a reasoner capable of doing backwards reasoning

Backwards reasoning over the same example

- bakcwards reasoning often suitable for stuff in memory
- you need a reasoner capable of doing backwards reasoning
- i.e. a rule reasoner

Backwards reasoning over the same example

- bakcwards 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

Backwards reasoning over the same example

- bakcwards 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`

Backwards reasoning over the same example

- bakcwards 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");
```

- `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);
```

- `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 ...