

IN3060/4060 – Semantic Technologies – Spring 2021

Lecture 6: Introduction to Reasoning with RDF

Jieying Chen

19th February 2021



DEPARTMENT OF
INFORMATICS



UNIVERSITY OF
OSLO

Mandatory exercises

- Oblig 4 published after this lecture.
- Hand-in by Friday in two weeks (05/03/2021).

Today's Plan

- 1 Inference rules
- 2 RDFS Basics
- 3 Backwards and forwards reasoning
- 4 RDFS reasoning in Jena

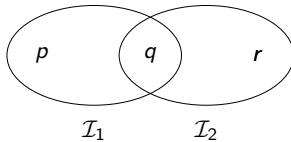
Outline

- 1 Inference rules
- 2 RDFS Basics
- 3 Backwards and forwards reasoning
- 4 RDFS reasoning in Jena

Model-theoretic semantics, a quick recap

We introduced *interpretations*:

- Idea: put all letters that are “true” into a set.
- Define: An *interpretation* \mathcal{I} is a set of letters.
- Letter p is true in interpretation \mathcal{I} if $p \in \mathcal{I}$.
- E.g., in $\mathcal{I}_1 = \{p, q\}$, p is true, but r is false.
- But in $\mathcal{I}_2 = \{q, r\}$, p is false, but r is true.



Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

- when a formula is *true* in an interpretation: $\mathcal{I} \models A$
- when a formula is a *tautology* (true in all interps.): $\models A$
- and when one formula *entails* another: $A \models B$.

Model-theoretic semantics is well-suited for

- studying the behaviour of a logic, since
- it is specified in terms of familiar mathematical objects, such as
 - sets of letters

Implementational disadvantages of model semantics

Model-theoretic semantics yields an unambiguous notion of entailment,

- But it isn't easy to read off from it what exactly is to be *implemented*.
- Much less does it provide an algorithmic means for *computing* it, that is
 - for actually *doing the reasoning*,
- In order to directly use the model-theoretic semantics,
 - in principle *all interpretations* would have to be considered.
 - But as there can be *infinitely many such interpretations*,
 - and an algorithm should terminate in *finite* time
 - this is not good.

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

- Syntactic methods operate only on the *form* of a statement, that is
- on its *concrete grammatical structure*,
- without recurring to interpretations,
- syntactic reasoning is, in other words, *computation*.

Interpretations still figure as the theoretical backdrop, as one typically

- strives to define syntactical methods that are *provably equivalent* to checking *all* interpretations

Syntactic reasoning easier to understand and use than model semantics

- we will show that first.

Inference rules

A calculus is usually formulated in terms of

- a set of *axioms* which are tautologies,
- and a set of *inference rules* for generating new statements.

The general form of an inference rule is:

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

- the P_i are *premises*
- and P is the *conclusion*.

An inference rule may have,

- any number of premises (typically one or two),
- but only one conclusion.

Where \models is the entailment relation, \vdash is the inference relation. We write $\Gamma \vdash P$ if we can deduce P from the assumptions Γ .

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

- One proves that,
 - every conclusion P derivable in the calculus from a set of premises Γ , is true in *all interpretations that satisfy* Γ . ($\Gamma \vdash P \Rightarrow \Gamma \models P$)
 - and conversely that every statement P entailed by Γ -interpretations is *derivable* in the calculus when the elements of Γ are used as premises. ($\Gamma \models P \Rightarrow \Gamma \vdash P$)

We say that the calculus is

- *sound* wrt the semantics, if (I) holds, and
- *complete* wrt the semantics, if (II) holds.

Inference rules in propositional logic

(Part of) Natural deduction calculus for propositional logic:

$$\frac{A \quad (A \rightarrow B)}{B} \rightarrow E$$

$$\frac{(A \wedge B)}{A} \wedge E_l \quad \frac{(A \wedge B)}{B} \wedge E_r \quad \frac{A \quad B}{(A \wedge B)} \wedge I$$

Inference for RDF

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*.

From this point of view a rule

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

may be read as an instruction;

- "If P_1, \dots, P_n are all in the store, *add* P to the store."

Outline

- 1 Inference rules
- 2 RDFS Basics
- 3 Backwards and forwards reasoning
- 4 RDFS reasoning in Jena

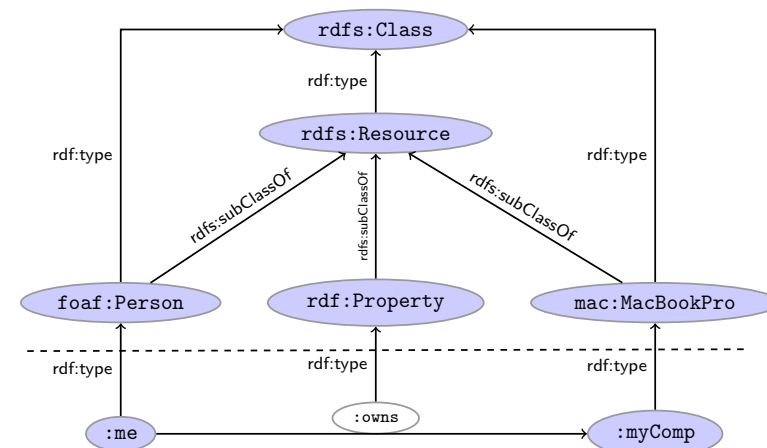
RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
 - `rdfs:` `http://www.w3.org/2000/01/rdf-schema#`
- Originally thought of as a “schema language” like XML Schema.
- Actually it isn't – doesn't describe “valid” RDF graphs.
- Comes with some inference rules
 - Allows to derive new triples mechanically.
- A very simple *modeling language*
- and (for our purposes) a subset of OWL.

RDF Schema concepts

- RDFS adds the concept of “classes” which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
 - `rdfs:Resource`: The class of resources, everything.
 - `rdfs:Class`: The class of classes.
 - `rd:Property`: The class of properties (from `rd:Property`).
- Defined properties:
 - `rd:type`: relate resources to classes they are members of.
 - `rd:domain`: The domain of a relation.
 - `rd:range`: The range of a relation.
 - `rd:subClassOf`: Class inclusion.
 - `rd:subPropertyOf`: Property inclusion.

Example



Intuition: Classes as Sets

- We can think of an `rdfs:Class` as denoting a *set* of Resources.
- Not quite correct, but OK for intuition.

RDFS	Set Theory
$A \text{ rdfs:type rdfs:Class}$	A is a set of resources
$x \text{ rdfs:type } A$	$x \in A$
$A \text{ rdfs:subClassOf } B$	$A \subseteq B$

Simple Entailment Rules

- Entailment with blank nodes and literals
- Without RDFS and RDF axioms
- $\frac{A \ R \ B \ .}{A \ R \ _ : x \ .} \text{ se1}$ $\frac{A \ R \ B \ .}{_ : x \ R \ B \ .} \text{ se2}$
- Where `_ : x` is a blank node identifier, that either
 - has not been used before in the graph, or
 - has been used, but for the same URI/Literal/Blank node.
 - `_ : x` represents `B` in `se1` and `A` in `se2`.

Simple Entailment Example

- Let's create the RDF-graph with the two triples:
 - 1 `:me :owns :myComp .`
 - 2 `:myComp rdfs:type mac:MacBookPro .`

Using **se1** on triple 1, it entails: `:me :owns _ : x`

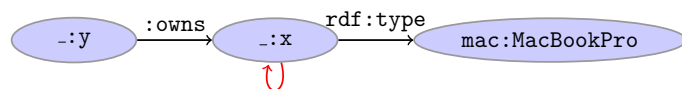
`_ : x` is not used previously

Using **se2** on triple 2 it entails: `_ : x rdfs:type mac:MacBookPro`

`_ : x` refers to the same URI

Using **se2** the inferred triple `:me :owns _ : x`

entails `_ : y :owns _ : x` where `_ : y` refers to the new URI



We can *not* infer `_ : x :owns _ : x` because `_ : x` was used for another URI.

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

- I. *Type propagation*:
 - "The 2CV *is a car*, and all cars *are motorised vehicles*, so..."
- II. *Property inheritance*:
 - "Steve *lectures at lfi*, and anyone who does so *is employed by lfi*, so..."
- III. *Domain and range reasoning*:
 - "Everything someone *has written* is a *document*. Alan *has written* 'Computing Machinery and Intelligence', therefore..."
 - "All *fathers* of people are *males*. James is the *father* of Karl, therefore..."

Type propagation with `rdfs:subClassOf`

The type propagation rules apply

- to combinations of `rdf:type`, `rdfs:subClassOf` and `rdfs:Class`,
- and trigger *recursive inheritance* in a *class taxonomy*.

Type propagation rules:

- *Members of subclasses*

$$\frac{A \text{ rdfs:subClassOf } B . \quad x \text{ rdf:type } A .}{x \text{ rdf:type } B .} \text{ rdfs9}$$

- *Reflexivity of sub-class relation*

$$\frac{A \text{ rdf:type rdfs:Class .}}{A \text{ rdfs:subClassOf } A .} \text{ rdfs10}$$

- *Transitivity of sub-class relation*

$$\frac{A \text{ rdfs:subClassOf } B . \quad B \text{ rdfs:subClassOf } C .}{A \text{ rdfs:subClassOf } C .} \text{ rdfs11}$$

Set Theory Analogy

- *Members of subclasses*

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

$$\frac{A \subseteq B \quad x \in A}{x \in B}$$

- *Reflexivity of sub-class relation*

$$\frac{A \text{ rdf:type rdfs:Class .}}{A \text{ rdfs:subClassOf } A .} \quad \frac{A \text{ is a set}}{A \subseteq A}$$

- *Transitivity of sub-class relation*

$$\frac{A \text{ rdfs:subClassOf } B . \quad B \text{ rdfs:subClassOf } C .}{A \text{ rdfs:subClassOf } C .}$$

$$\frac{A \subseteq B \quad B \subseteq C}{A \subseteq C}$$

Example

RDFS/RDF knowledge base:

```
ex:Vertebrate rdf:type rdfs:Class .
ex:Mammal rdf:type rdfs:Class .
ex:KillerWhale rdf:type rdfs:Class .

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

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

Inferred triples:

```
ex:Keiko rdf:type ex:Mammal . (rdfs9)
ex:Keiko rdf:type ex:Vertebrate . (rdfs9)
ex:KillerWhale rdfs:subClassOf ex:Vertebrate . (rdfs11)
ex:Mammal rdfs:subClassOf ex:Mammal . (rdfs10)
(... and also for the other classes)
```

A typical taxonomy

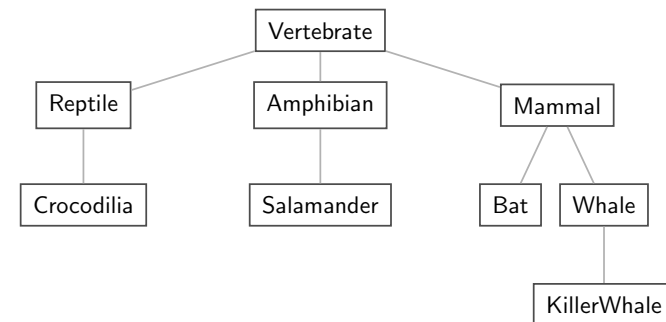


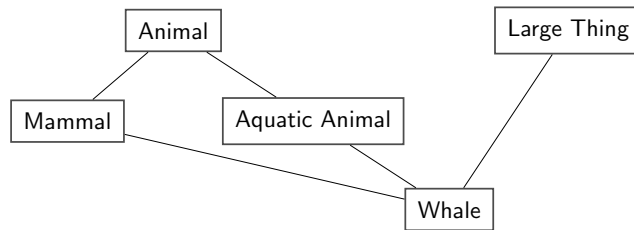
Figure: A typical taxonomy

Multiple Inheritance

- A set is a subset of many other sets:

$$\{2, 3\} \subseteq \{1, 2, 3\} \quad \{2, 3\} \subseteq \{2, 3, 4\} \quad \{2, 3\} \subseteq \mathbb{N} \quad \{2, 3\} \subseteq \mathbb{P}$$

- Similarly, a class is usually a subclass of many other classes.



- This is usually not called a *taxonomy*, but it's no problem for RDFS.

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

- `rdfs:subPropertyOf`,
- `rdf:type`, and
- `rdf:Property`

Rules for property reasoning:

- *Transitivity:*

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

- *Reflexivity:*

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

- *Property transfer:*

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

Intuition: Properties as Relations

- If an `rdfs:Class` is like a set of resources...
- ... then an `rdf:Property` is like a relation on resources.
- Remember: not quite correct, but OK for intuition.

RDFS	Set Theory
$r \text{ rdf:type } \text{rdf:Property}$	$r \text{ is a relation on resources}$
$x \text{ r } y$	$\langle x, y \rangle \in r$
$r \text{ rdfs:subPropertyOf } s$	$r \subseteq s$

- Rules:

$$\frac{p \subseteq q \quad q \subseteq r}{p \subseteq r} \quad \frac{p \text{ a relation}}{p \subseteq p} \quad \frac{p \subseteq q \quad \langle u, v \rangle \in p}{\langle u, v \rangle \in q}$$

Example I: Harmonizing terminology

Integrating data from multiple sources in general requires:

- Harmonisation of the data under a common vocabulary.

The aim is to

- make similar data answer to *the same standardised queries*,
- thus making queries *independent of the terminology of the sources*.

For instance:

- Suppose that a legacy bibliography system *S* uses `:author`, where
- another system *T* uses `:writer`.

And suppose we wish to integrate *S* and *T* under a common scheme,

- for instance Dublin Core.

Solution

From Ontology:

```

:writer rdf:type rdf:Property .
:author rdf:type rdf:Property .
:author rdfs:subPropertyOf dcterms:creator .
:writer rdfs:subPropertyOf dcterms:creator .

```

And Facts:

```

ex:knausgård :writer ex:minKamp .
ex:hamsun :author ex:sult .

```

Infer:

```

ex:knausgård dcterms:creator ex:minKamp .
ex:hamsun dcterms:creator ex:sult .

```

Consequences

- Any individual for which `:author` or `:writer` is defined,
- will have the same value for the `dcterms:creator` property.
- The work of integrating the data is thus done by the reasoning engine,
- instead of by a manual editing process.
- Legacy applications that use e.g. `author` can operate unmodified.

Example II: Keeping track of employees

Large organizations (e.g. universities) offer different kinds of contracts;

- for tenured positions (professors, assisting professors, lecturers),
- for research associates (Post Docs),
- for PhD students,
- for subcontracting.

Employer/employee information can be read off from properties such as:

- `:profAt` (*professorship at*),
- `:tenAt` (*tenure at*),
- `:conTo` (*contracts to*),
- `:funBy` (*is funded by*),
- `:recSchol` (*receives scholarship from*).

Organising the properties

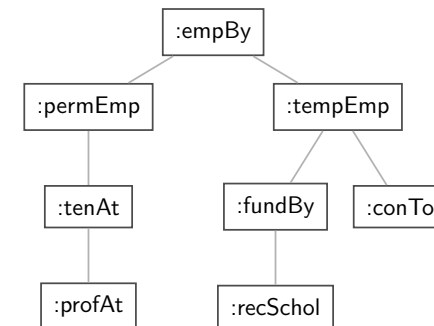


Figure: A hierarchy of employment relations

- Note: doesn't have to be tree-shaped.

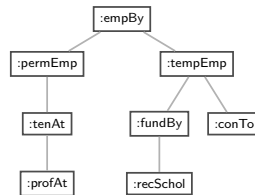
Querying the inferred model

Formalising the tree:

```
:profAt rdf:type rdfs:Property .
:tenAt rdf:type rdfs:Property .
:profAt rdfs:subPropertyOf :tenAt
..... and so forth.
```

Given a data set such as:

```
:Martin :profAt :Ui0 .
:Ole :fundBy :Ui0 .
:Steve :conTo :OLF .
:Trond :recSchol :BI .
:Jenny :tenAt :SSB .
```



cont.

We may now query on different levels of abstraction :

Temporary employees

```
SELECT ?emp WHERE {?emp :tempEmp _:x .}
→ Ole, Steve, Trond
```

Permanent employees

```
SELECT ?emp WHERE {?emp :permEmp _:x .}
→ Martin, Jenny
```

All employees

```
SELECT ?emp WHERE {?emp :empBy _:x .}
→ Martin, Jenny, Ole, Steve, Trond
```

Third pattern: Typing data based on their use

Triggered by combinations of

- `rdfs:range`
- `rdfs:domain`
- `rdf:type`

Rules for domain and range reasoning :

- *Typing first coordinates:*

$$\frac{p \text{ rdfs:domain } A \quad x \text{ p } y \quad \text{rdfs2}}{x \text{ rdf:type } A \quad .}$$

- *Typing second coordinates:*

$$\frac{p \text{ rdfs:range } B \quad x \text{ p } y \quad \text{rdfs3}}{y \text{ rdf:type } B \quad .}$$

Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is *used*.
- `rdfs:domain` types the *possible subjects* of these triples,
- whereas `rdfs:range` types the *possible objects*,
- When we assert that property `p` has domain `C`, we are saying
 - that whatever resource is linked to anything by `p`
 - this resource must be of type `C`.

Domain and Range of Relations

- Given a relation R from A to B ($R \subseteq A \times B$)
- The *domain* of R is the set of all x with $xR \dots$:

$$\text{dom } R = \{x \in A \mid xRy \text{ for some } y \in B\}$$

- The *range* of R is the set of all y with $\dots R y$:

$$\text{rg } R = \{y \in B \mid xRy \text{ for some } x \in A\}$$

- Example:

- $R = \{ \langle 1, \triangle \rangle, \langle 1, \square \rangle, \langle 2, \diamond \rangle \}$
- $\text{dom } R = \{1, 2\}$
- $\text{rg } R = \{ \triangle, \square, \diamond \}$

Set intuitions for `rdfs:domain` and `rdfs:range`

- If an `rdfs:Class` is like a set of resources and an `rdf:Property` is like a relation on resources...

RDFS	Set Theory
$r \text{ rdfs:domain } A$	$\text{dom } r \subseteq A$
$r \text{ rdfs:range } B$	$\text{rg } r \subseteq B$

- Rules:

$$\frac{\text{dom } p \subseteq A \quad \langle x, y \rangle \in p}{x \in A}$$

$$\frac{\text{rg } p \subseteq B \quad \langle x, y \rangle \in p}{y \in B}$$

Example I: Combining domain, range and `subClassOf`

Suppose we have a class hierarchy that includes:

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

and a property `:conductor` whose domain and range are:

```
:conductor rdfs:domain :SymphonyOrchestra .
```

```
:conductor rdfs:range :Person .
```

Now, if we assert

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

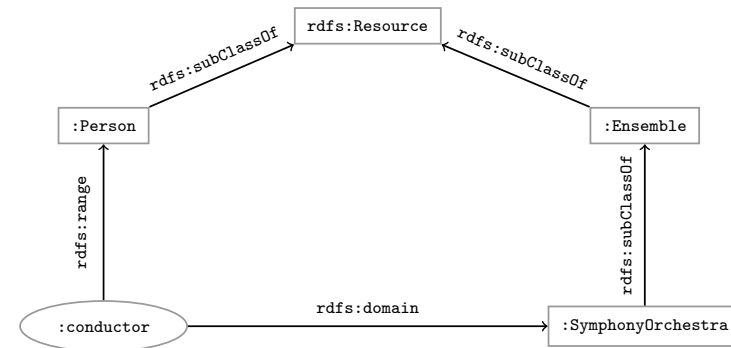
we may infer;

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

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

```
:Petrenko rdf:type :Person .
```

Conductors and ensembles



Example II: Filtering information based on use

Consider once more the dataset:

```
:Martin :profAt :UiO .
:Ole :fundBy :UiO .
:Steve :conTo :OLF .
:Trond :recSchol :BI .
:Jenny :tenAt :SSB .
```

and suppose we wish to filter out everyone but the freelancers:

- State that only freelancers :conTo an organisation,
- i.e. introduce a class :Freelancer,
- and declare it to be the domain of :conTo:


```
:Freelancer rdf:type rdfs:Class .
:conTo rdfs:domain :Freelancer .
```

Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

```
:conTo rdfs:domain :Freelancer .
:Steve :conTo :OLF .
:Steve rdf:type :Freelancer rdfs2
```

and may be used as a type in SPARQL (reasoner presupposed):

Finding the freelancers

```
SELECT ?freelancer WHERE {
  ?freelancer rdf:type :Freelancer .
}
```

RDFS axiomatic triples (excerpt)

Some triples are *axioms*: they can always be added to the knowledge base.

- Only resources have types:


```
rdf:type rdfs:domain rdfs:Resource .
```
- types are classes:


```
rdf:type rdfs:range rdfs:Class .
```
- Ranges apply only to properties:


```
rdfs:range rdfs:domain rdf:Property .
```
- Ranges are classes:


```
rdfs:range rdfs:range rdfs:Class .
```
- Only properties have subproperties:


```
rdfs:subPropertyOf rdfs:domain rdf:Property .
```
- Only classes have subclasses:


```
rdfs:subClassOf rdfs:domain rdfs:Class .
```
- ... (another 30 or so)

Using the Axiomatic Triples

- From the statement


```
:conductor rdfs:range :Person
```
- We can derive:
 - :conductor rdf:type rdf:Property
 - :Person rdf:type rdfs:Class
 - :conductor rdf:type rdfs:Resource
 - rdf:Property rdf:type rdfs:Class
 - :Person rdfs:type rdfs:Resource
 - rdfs:Class rdfs:type rdfs:Class
 - ...
- In OWL, there are some simplification which make this superfluous.

Writing proofs

When writing proofs, we:

- write one triple per line,
- enumerate the lines,
- write the rule name along with the line numbers corresponding to the assumptions,
- introduce triples from the knowledge base with the rule name *P*.
- E.g. given the knowledge base:

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

- We write:

- 1 :OsloPhilharmonic :conductor :Petrenko . - P
- 2 :conductor rdfs:domain :SymphonyOrchestra . - P
- 3 :OsloPhilharmonic rdf:type :SymphonyOrchestra . - rdfs3, 1, 2
- 4 :SymphonyOrchestra rdfs:subClassOf :Ensemble . - P
- 5 :OsloPhilharmonic rdf:type :Ensemble . - rdfs9, 3, 4

Outline

- 1 Inference rules
- 2 RDFS Basics
- 3 Backwards and forwards reasoning
- 4 RDFS reasoning in Jena

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

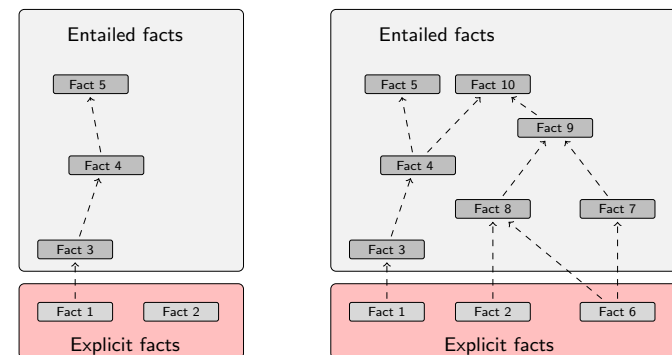


Figure: When a fact is added, all entailments are computed and stored.

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

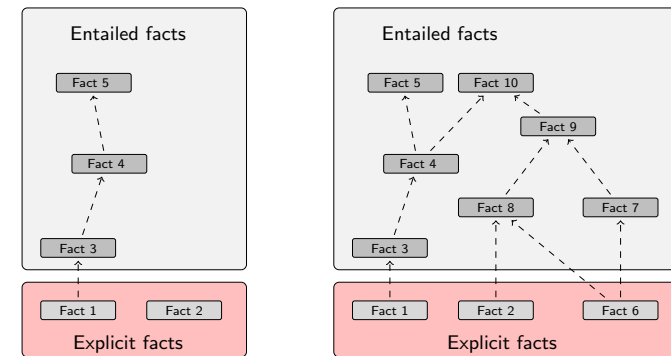


Figure: When a fact is added, all entailments are computed and stored.

Forward chaining and truth-maintenance

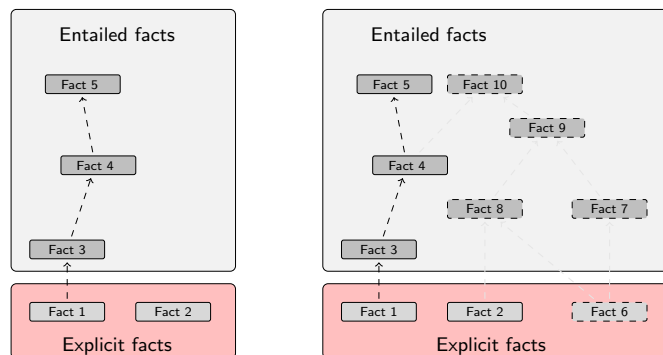


Figure: When a fact is removed, everything that comes with it must go too.

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

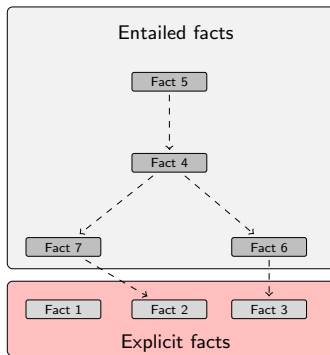


Figure: Backward chaining uses rules to expand queries.

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 .
  
```

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

Query:

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

Inferred triples:

```

?x rdf:type ex:Vertebrate .
?x rdf:type ex:Mammal . (rdfs9)
?x rdf:type ex:KillerWhale . (rdfs9) ⇒ ?x = ex:Keiko
?x rdf:type ex:Lion . (rdfs9) ⇒ ?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

- 1 Inference rules
- 2 RDFS Basics
- 3 Backwards and forwards reasoning
- 4 RDFS reasoning in Jena

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

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

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

Conclusion

- We have seen that by modelling knowledge using the URIs in the RDF and RDFS vocabularies (e.g. `rdf:type`, `rdfs:subClassOf`, `rdfs:range`), the computer can derive *new* triples, that follows from our original triples.
- The rules were very simple (e.g. if `x rdf:type A` and `A rdfs:subClassOf B` then `x rdf:type B`).
- However, note that even the most complex mathematical proofs can be broken down into equally simple steps.
- It is when we have large knowledge bases and we can apply thousands or millions of derivations that the reasoning becomes really interesting.
- Example of large ontology, BabelNet: <http://www.babelnet.org/>
- OWL will also allow us to express more complex statements and use more complex types of reasoning.

That's it for today!

Remember the oblig!