INF3580/4580 – Semantic Technologies – Spring 2017

Lecture 6: Introduction to Reasoning with RDF

Leif Harald Karlsen

20th February 2017

Department of
Informatics

University of
Oslo

Mandatory exercises

- Oblig 4 published after this lecture.
- Hand-in by Tuesday in two weeks.
- Exercises mostly from this week's lecture, but one from next week's lecture, Reasoning with Jena.

Today's Plan

1. Inference rules
2. RDFS Basics
3. Open world semantics

Outline

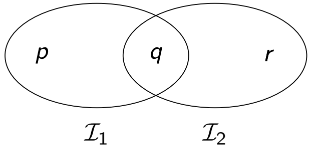
1. Inference rules
2. RDFS Basics
3. Open world semantics

Model-theoretic semantics, a quick recap

The previous lecture introduced a “model-theoretic” semantics for Propositional Logic.

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

Preview: Model Semantics for RDF

- We will look at semantics for RDF in two weeks.
- Interpretations will consist of
  - a set $\mathcal{D}$ of resources (possibly infinite),
  - a function mapping each URI to an object in $\mathcal{D}$,
  - relations on $\mathcal{D}$ giving meaning for each property.
- Everything else will be defined in terms of these interpretations.
- Entailment of RDF graphs, etc.
- Remember: interpretations for Propositional Logic could be listed in truth tables.
  - Only 2^n possibilities for n letters.
- Not possible for RDF:
  - ∞ many different interpretations

Implementational disadvantages of model semantics

Model-theoretic semantics yields an unambigous 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 are always *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:

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

- 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,
    - I. every conclusion P derivable in the calculus from a set of premises Γ, is true in *all interpretations that satisfy* Γ. ($\Gamma \vdash P \Rightarrow \Gamma \models P$)
    - II. 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 dedcution calclulus for propositional logic:

$$\frac{A \qquad (A \to B)}{B} \to E$$

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

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF store (broadly construed),
- on the basis of the triples *already in it*.

From this point of view a rule

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

may be read as an instruction;

- ”If $P_1, \ldots, P_n$ are all in the store, *add* P to the store.”

Outline

1. Inference rules
2. RDFS Basics
3. Open world semantics

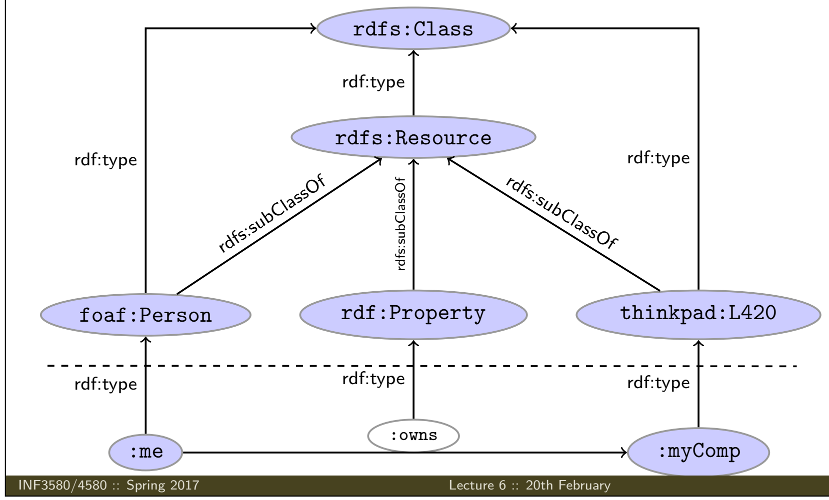
RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`
- Originally though 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.
  - `rdf:Property`: The class of properties (from `rdf`).
- Defined properties:
  - `rdf:type`: relate resources to classes they are members of.
  - `rdfs:domain`: The domain of a relation.
  - `rdfs:range`: The range of a relation.
  - `rdfs:subClassOf`: Class inclusion.
  - `rdfs: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 `rdf:type rdfs:Class`	A is a set of resources
x `rdf:type` A	$x \in A$
A `rdfs:subClassOf` B	$A \subseteq B$

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:*
   - “Steve *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*. Alan *has written* a book, therefore. . . ”
   - “All *father*s of people are *male*s. 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*

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

- *Reflexivity of sub-class relation*

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

- *Transitivity of sub-class relation*

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

Set Theory Analogy

- *Members of subclasses*

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

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

- *Reflexivity of sub-class relation*

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

- *Transitivity of sub-class relation*

$$\frac{\texttt{A rdfs:subClassOf B .}\qquad \texttt{B rdfs:subClassOf C .}}{\texttt{A rdfs:subClassOf C .}}$$

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

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

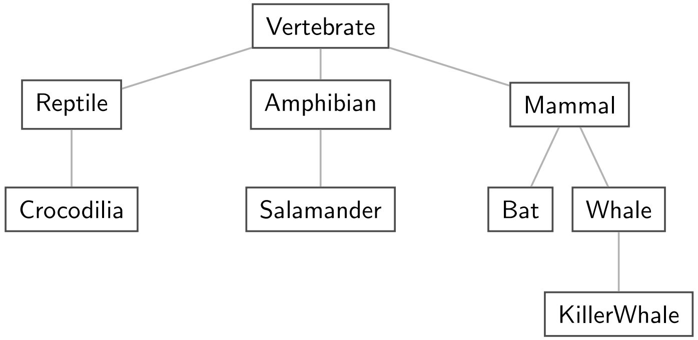


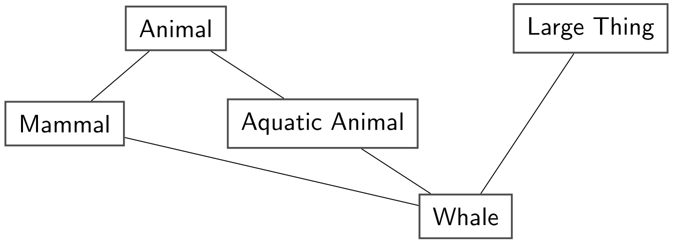
Figure: A typical taxonomy

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:*

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

- *Reflexivity:*

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

- *Property transfer:*

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

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 `rdf:type rdf:Property`	r is a relation on resources
$x\ r\ y$	$\langle x, y\rangle \in r$
r `rdfs:subPropertyOf` s	$r \subseteq s$

- Rules:

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

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

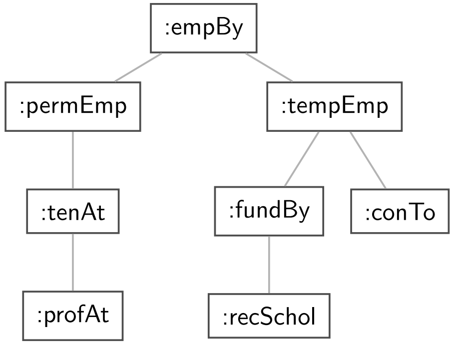


Figure: A hierarchy of employment relations

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

Querying the inferred model

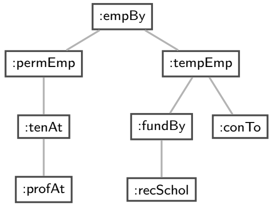
Formalising the tree:

```
:profAt rdf:type rdfs:Property .
:tenAt rdf:type rdfs:Property .
:profAt rdfs:subPropertyOf :tenAt
```

..... and so forth.

Given a data set such as:

```
:Arild :profAt :UiO .
:Audun :fundBy :UiO .
: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 .}`
→ *Audun, Steve, Trond*

Permanent employees

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

All employees

`SELECT ?emp WHERE {?emp :empBy _:x .}`
→ *Arild, Jenny, Audun, 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:*

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

- *Typing second coordinates:*

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

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 is linked to anything by `p`
  - must be an object of type `C`,
  - therefore an application of `p` suffices to type that resource.

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 $x\,R \cdots$:

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

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

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

- Example:
  - $R = \{\langle 1, \triangle\rangle, \langle 1, \square\rangle, \langle 2, \lozenge\rangle\}$
  - $\operatorname{dom} R = \{1, 2\}$
  - $\operatorname{rg} R = \{\triangle, \square, \lozenge\}$

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 `rdfs:domain` A	$\mathrm{dom}\, r \subseteq A$
r `rdfs:range` B	$\mathrm{rg}\, r \subseteq B$

- Rules:

$$\frac{\mathrm{dom}\, p \subseteq A \qquad \langle x, y\rangle \in p}{x \in A}$$

$$\frac{\mathrm{rg}\, p \subseteq B \qquad \langle x, y\rangle \in p}{y \in B}$$

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 .
```

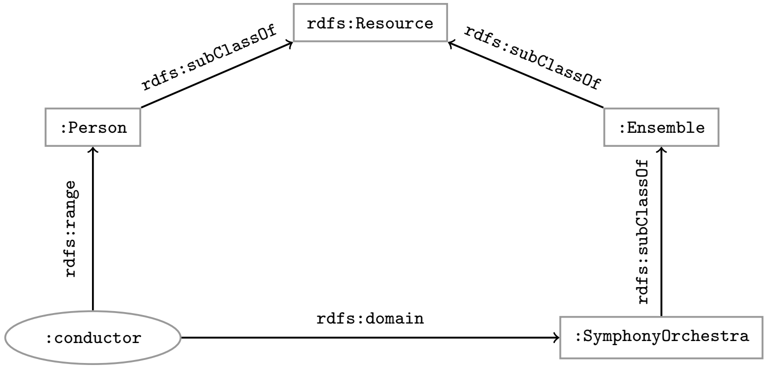
Now, if we assert

```
: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:

```
:Arild :profAt :UiO .
:Audun :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,

$$\frac{\text{:conTo rdfs:domain :Freelancer .} \qquad \text{:Steve :conTo :OLF .}}{\text{:Steve rdf:type :Freelancer}}\ \text{rdfs2}$$

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:
  ❶ `:OsloPhilharmonic :conductor :Petrenko .` – P
  ❷ `:conductor rdfs:domain :SymphonyOrchestra .` – P
  ❸ `:OsloPhilharmonic rdf:type :SymphonyOrchestra .` – rdfs3, 1, 2
  ❹ `:SymphonyOrchestra rdfs:subClassOf :Ensemble .` – P
  ❺ `:OsloPhilharmonic rdf:type :Ensemble .` – rdfs9, 3, 4

Outline

1. Inference rules
2. RDFS Basics
3. Open world semantics

Gentle RDFS

Recall that RDF *Schema* was conceived of as a schema language for RDF.

- However, the statements in an RDFS ontology *never trigger inconsistencies*.
- I.e. no amount of reasoning will lead to a "contradiction", "error", "non-valid document"
- Example: Say we have the following triples;

```
:isRecordedBy rdfs:range :Orchestra .
:Beethovens9th :isRecordedBy :Boston .
```

- Suppose now that `Boston` is *not* defined to be an `Orchestra`:
  - i.e., there is no triple `:Boston rdf:type :Orchestra .` in the data.
- in a standard relational database,
- it would follow that `:Boston` is *not* an `:Orchestra`,
- which contradicts the rule `rdfs7`:

$$\frac{\texttt{:isRecordedBy rdfs:range :Orchestra .} \qquad \texttt{:Beethovens9th :isRecordedBy :Boston .}}{\texttt{:Boston rdf:type :Orchestra .}}\ \text{rdfs7}$$

Contd.

Instead;

- RDFS infers *a new triple*.
- More specifically it *adds* `:Boston rdf:type :Orchestra .`
- which is precisely what `rdfs7` is designed to do.

This is *open world reasoning* in action:

- Instead of saying "I know that `:Boston` is not an `:Orchestra`",
- RDFS says "`:Boston` *is* an `:Orchestra`, I just didn't know it."
- RDFS will not signal an inconsistency,
- but rather just add the missing information

This is *the* most important difference between relational DBs and RDF.

Ramifications

This fact has two important consequences:

1. RDFS is useless for validation,
   - ... understood as sorting conformant from non-conformant documents,
   - since it never signals an inconsistency in the data,
   - it just goes along with anything,
   - and adds triples whenever they are inferred.
   - Note though, that validation functionality beyond RDFS is often implemented in RDFS reasoners.
2. RDFS has no notion of negation *at all*
   - For instance, the two triples

     `ex:Joe rdf:type ex:Smoker .,`
     `ex:Joe rdf:type ex:NonSmoker .`

     are not inconsistent.
     - (It is not possible to in RDFS to say that `ex:Smoker` and `ex:nonSmoker` are disjoint).

Expressive limitations of RDFS

Hence,

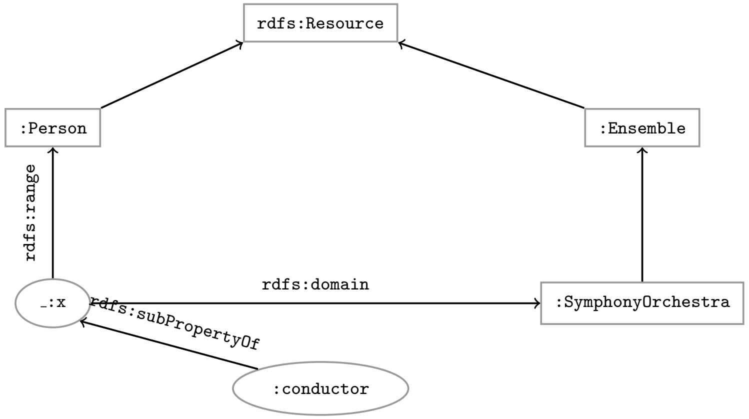
- RDFS cannot express inconsistencies,
- so *any* RDFS graph is consistent.

Therefore,

- RDFS supports no reasoning services that require consistency-checking.
- If consistency-checks are needed, one must turn to OWL.
- More about that in a few weeks.

A conspicuous non-pattern

Suppose we elaborate on our music example in the following way:



The incompleteness of RDFS

That is:

- We make `:conductor` a subproperty of `_:x`,
- `_:x` is a generic relation between people and orchestras,
- to be used whenever we want the associated restrictions.

We would then *want to be able* to reason as follows (names abbreviated):

① `:Oslo :cond :Abadi .` – P

② `:cond rdfs:subProp _:x .` – P

③ `:Oslo _:x :Abadi .` – rdfs7, 1, 2

④ `_:x rdfs:domain :Person .` – P

⑤ `:Abadi rdfs:type :Person .` – rdfs2, 3, 4

Contd.

- However, we cannot use `rdfs2` and `rdfs7` in this way,
- since it requires putting a blank in predicate position,
- which is not legitimate RDF.
- Hence, the conclusion is not derivable.

Nevertheless,

- this really *is a semantically valid inference*,
- ... you are hereby encouraged to check this for yourself,
- whence the RDFS rules are *incomplete* wrt. RDFS semantics.

Assessing the situation

RDFS reasoners usually implement only the standardised incomplete rules, so

- they do not guarantee complete reasoning.

Better therefore;

- if all you need is the three RDFS reasoning patterns,
- to use OWL and OWL reasoners instead.

Unless, of course

- you need to talk about properties and classes as objects,
- that is, you need the meta-modelling facilities of RDFS,
- but people rarely do.

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, SNOMED: `http://browser.ihtsdotools.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!