# INF3580 – Semantic Technologies – Spring 2010

## Lecture 6: RDFS and RDFS design patterns

Audun Stolpe

2nd March 2010

Department of Informatics
University of Oslo

## Today's Plan

1. Inference rules
2. RDFS basics
3. RDFS design patterns
4. Domains, ranges and open worlds





## Outline

1. Inference rules
2. RDFS basics
3. RDFS design patterns
4. Domains, ranges and open worlds





## Model-theoretic semantics, a quick recap

The previous lecture introduced a model-theoretic semantics for RDF:

- we specified in a mathematically precise way
  - when a triple is true according to a given graph,
  - and when one graph is entailed by another.

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
  - functions,
  - variables, and
  - relations.

## 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 models would have to be considered.
  - But as there are always infinitely many such interpretations,
  - and an algorithm is a finite object
  - this is impossible.





## 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, calculation.

Interpretations still figure as the theoretical backdrop, as one typically

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





## Soundness and completeness

Semantics and calculus are typically made to work like chopsticks:

- One proves that,
  I. every conclusion derivable in the calculus from a set of premises $A$, is true in all models that satisfy $A$
  II. and conversely that every statement entailed by $A$-models is derivable in the calculus when the elements of $A$ are used as premises.

We say that the calculus is

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





## Inference rules

A calculus is usually formulated in terms of

- a set of axioms that represent unquestioned postulates,
- 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 (obviously).

## Describing models

Given soundness and completeness of a calculus wrt a semantics:

- the axioms are true in all models (aka valid in the class), and
- $P$ is entailed by all models satisfying $P_1, \ldots, P_n$

In other words,

- the calculus may be considered a description of the models:
  - $P_1, \ldots, P_n$-models in which $P$ is false are, for instance, disallowed.
- The calculus thus fixes the basic structure of the world
  - as one chooses to see it





## RDFS axiomatic triples (excerpt)

### RDFS axiomatics

- Only resources have types:

  `rdf:type rdfs:domain rdfs:Resource .`
- Domains apply only to properties:

  `rdfs:domain rdfs:domain rdf:Property .`
- Ranges apply only to properties:

  `rdfs:range rdfs:domain rdf:Property .`
- Only properties are subproperties:

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

  `rdfs:subClassOf rdfs:domain rdfs:Class .`





## Inference in the concrete

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"





## Studying RDFS through inference rules

In this lecture we shall

- study RDFS by example,
- by switching from the semantic point of view,
- to the RDFS inference rules given in the RDFS specification.

All these rules are sound wrt to RDFS semantics, hence

- They will never produce incorrect conclusions

However, as it happens they are not complete, that is

- the calculus will not give you all semantically licensed conclusions.

## RDFS incompleteness

The RDFS incompleteness is manifest in certain cases of

- domain and range reasoning.

It is important to be aware of which ones and why, because

- they are quite innocuous looking reasoning patterns,
- that it would otherwise be natural to trust.

Nevertheless, RDFS reasoners usually do not implement them, so

- one may easily end up spending hours trying to fix,
- something that isn't really broken (just badly specified).

We shall see in lecture 7 that these flaws do not carry over to OWL.





## Outline







## RDFS in a nutshell

RDFS has often been thought of as a *schema language* for RDF.

- For reasons I shall come back to, this is a habit to suspend,
  - at least if a RDF schema is thought of in analogy to, say, a DTD,
  - as something that limits the set of valid documents.
- RDFS is best thought of as simple system for reasoning about types,
- that is, as a simple ontology language.

RDFS supports three principal kinds of reasoning pattern:

I. Type propagation:
   - "The beetle 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:
   - "Only people have birth certificates. Martin has one, therefore ..."





## Concepts in the RDFS datamodel

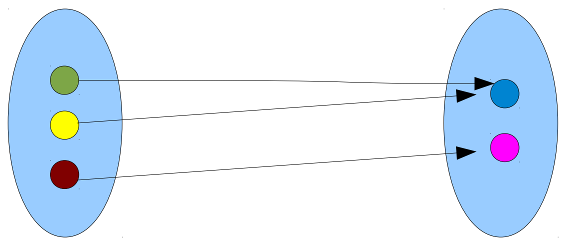
### Corresponding vocabulary items

- **RDFS classes:**
  - `rdfs:Resource`: The class of resources, everything.
  - `rdfs:Class`: The class of classes.
  - `rdf:Property`: The class of properties (from `rdf`)
- **RDFS properties**
  - `rdfs:domain`: The domain of a relation.
  - `rdfs:range`: The range of a relation.
  - `rdfs:subClassOf`: Concept inclusion.
  - `rdfs:subPropertyOf`: Property inclusion.

## Talking about sets and relations

Thus RDFS is about sets and relations:

- Nodes are grouped into `rdfs:Classes`.
- One class may be an `rdfs:subClassOf` another.
- Edges are grouped into `rdf:Properties`.
- Properties may be given `rdfs:domains` and `rdfs:ranges`







## contd.

Finally,

- a property may be a `rdfs:subPropertyOf` another.

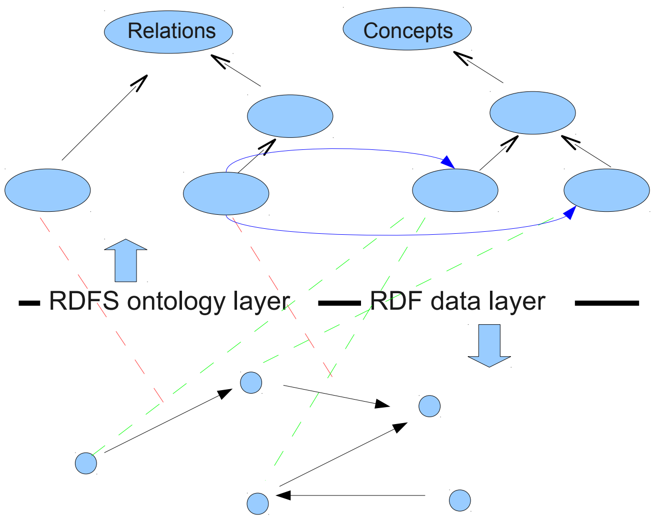
Stated plainly RDFS is thus

- a simple language for defining class and property taxonomies,
- that is, for defining simple hierarchies of concepts and relations,
- with the ability to interconnect the two by domains and ranges.





## An RDFS knowledge base








## Things to note

This is not an entirely accurate depiction, since

- there is no clear distinction between data and ontology in RDFS.
- This is due to the non-extensional semantics of RDFS:

Remember;

- Properties may act both as objects and relations, and
- classes may act both as objects and sets,
- in effect blurring the line.

Nevertheless, this tends to be a convenient way to think about it.

## Outline







## First pattern: 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:**

- Membership abstraction:

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

- Reflexivity of subsumption:

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

- Transitivity of subsumption:

$$\frac{\texttt{u rdfs:subClassOf v .} \qquad \texttt{v rdfs:subClassOf x .}}{\texttt{u rdfs:subClassOf x .}}\ \text{rdfs11}$$





## Example

**RDFS ontology:**

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

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

**RDF facts:**

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

**Inferred triples:**

```
ex:Keiko rdf:type ex:Mammal .
ex:Keiko rdf:type ex:Vertebrate .
```

and, `ex:Keiko rdf:type rdfs:Resource .` (from the axiomatic triples).





## A typical taxonomy

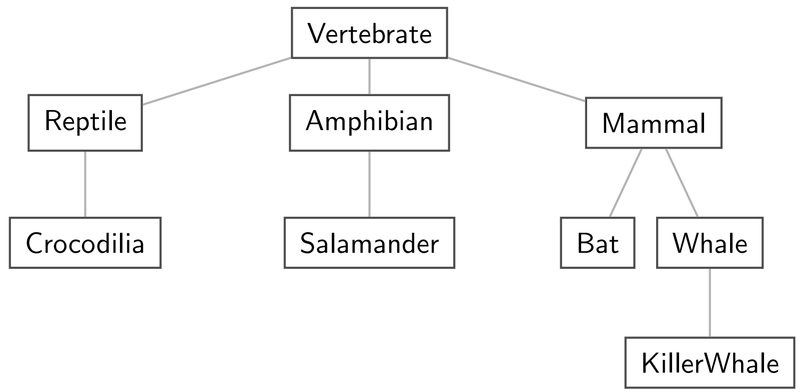



Figure: A typical taxonomy

## Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depend on certain combinations of

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

**Rules for property reasoning:**

- Transitivity:

$$\frac{\texttt{u rdfs:subPropertyOf v .} \quad \texttt{v rdfs:subPropertyOf x .}}{\texttt{u rdfs:subPropertyOf x .}}\ \text{rdfs5}$$

- Reflexivity:

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

- Property transfer:

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





## 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

**RDFS Ontology:**

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

**Effects:**

- 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 RDFS 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

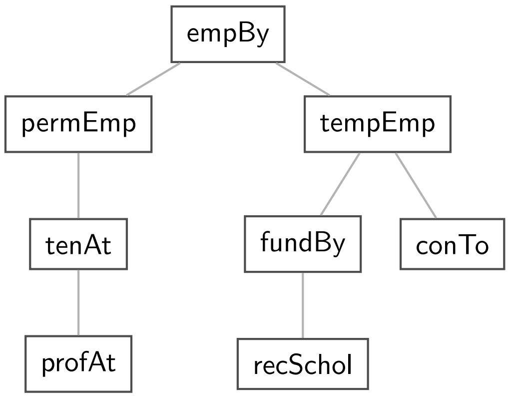



Figure: A hierarchy of employment relations





## 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 .
:Martin :conTo :OLF .
:Trond :recSchol :BI .
:Jenny :tenAt :SSB .
```





## cont.

**We may now query on different levels of abstraction :**

Aggregating employment relations

```
SELECT ?temp ?perm ?all WHERE {
    ?temp :tempEmp _:x .
    ?perm :permEmp _:y .
    ?all :empBy _:z .
}
```

**And get different aggregates in return:**

| all | perm | temp |
|---|---|---|
| Arild | Arild | |
| Jenny | Jenny | |
| Martin | | Martin |
| Audun | | Audun |
| Trond | | Trond |





## Third pattern: Typing data based on their use

Triggered by combinations of

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

**Rules for damain and range reasoning :**

- Typing first coordinates:

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

- Typing second coordinates:

$$\frac{\texttt{p rdfs:range u .} \quad \texttt{x p y .}}{\texttt{y rdf:type u .}}\ \text{rdfs2}$$

## Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is used.
- `rdfs:domain` types the possible 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`,
  - wherefore an application of `p` suffices to type that resource.





## Example I: Combining domain, range and subClassOf

Suppose we have a class tree 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 :Saraste .
```

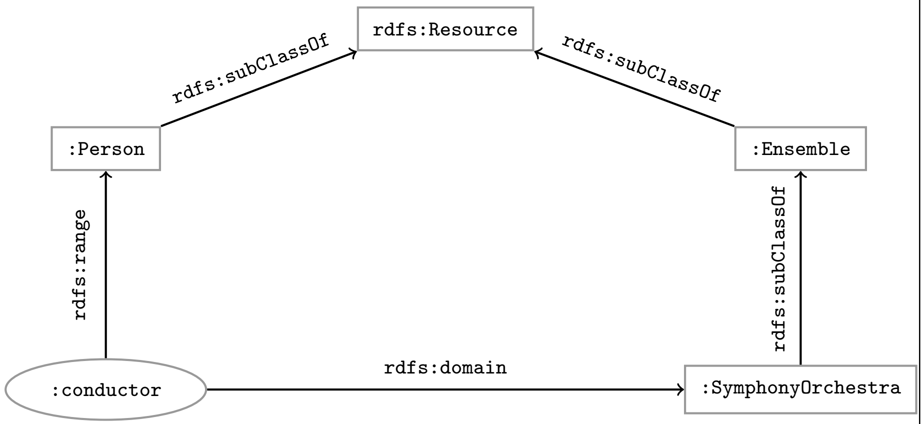
we may infer;

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





## Conductors and ensembles








## Example II: Filtering information based on use

Consider once more the dataset:

```
:Arild :profAt :UiO .
:Audun :fundBy :UiO .
:Martin :conTo :OLF .
:Trond :recSchol :BI .
:Jenny :tenAt :SSB .
```

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

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

```
:freelancer rdf:type rdfs:Class .
:contractsTo rdfs:domain :Freelancer .
```

## Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

:contractsTo rdfs:domain :Freelancer .    :Martin :contractsTo :OLF .
——————————————————————————— rdfs2
:Martin rdf:type :Freelancer

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

**Finding the freelancers**

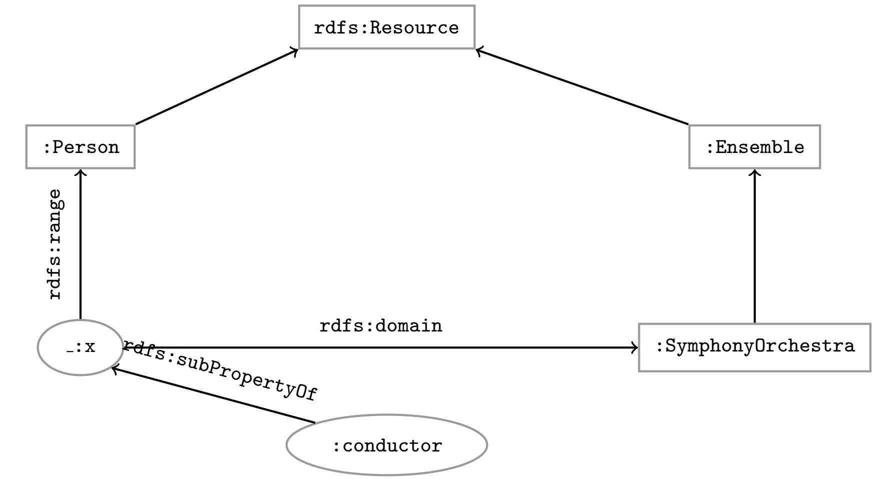
```
SELECT ?freelancer WHERE {
    ?freelancer rdf:type :Freelancer .
}
```





## 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):

```
:Oslo :vis :Abadi .    :vis rdfs:subProp _:x .
——————————————————————————————————————————— rdfs7
       :Oslo _:x :Abadi .                     _:x rdfs:domain :Person
rdfs2 ———————————————————————————————————————————————————————————————
                    :Abadi rdfs:type :Person
```





## 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.





## Outline







## 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.
- This is due to the open world semantics of RDFS.
- Example: Say we have the following triples;

```
:isRecordedBy rdfs:range :Orchestra .
:Turangalila :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.
- Then if the closed world assumption were adopted,
- it would follow that :Boston is not an :Orchestra,
- which contradicts the rule rdfs7:

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





## 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, therefore
- but rather just add the missing information

## 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,
   - It is *in this respect* more like a database schema,
   - which declares what joins are possible,
   - but makes no statement about the validity of the joined data.
   - 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:Martin rdf:type ex:Smoker .,
     ex:Martin 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 the next lecture.





## Supplementary reading

- For RDFS design patterns:

  *Semantic Web for the Working Ontologist.*
  Allemang, Hendler.
  Morgan Kaufmann 2008
  *Read chapter 6.*

- For RDFS semantics:

  *Read chapter 3.*