

# INF3580 – Semantic Technologies – Spring 2011

## Lecture 6: Introduction to Reasoning with RDF

Martin Giese

1st March 2010



DEPARTMENT OF  
INFORMATICS



UNIVERSITY OF  
OSLO

## Today's Plan

- 1 Inference rules
- 2 RDFS Basics
- 3 Domains, ranges and open worlds

## Outline

- 1 Inference rules
- 2 RDFS Basics
- 3 Domains, ranges and open worlds

## Model-theoretic semantics, a quick recap

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

- we specified in a mathematically precise way
  - when a formula is **true** in an interpretation,
  - when a formula is a “**tautology**” (true in all interps.)
  - and when one formula **entails** 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
  - **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:
  - $\infty$  many different interpretations

## 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 are always **infinitely many such interpretations**,
  - and an algorithm must terminate in **finite** time
  - 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* interpretations

Syntactic reasoning easier to understand and use than model semantics

- we will show that first!

## Soundness and completeness

Semantics and calculus are typically made to work like chopsticks:

- One proves that,
  - every conclusion derivable in the calculus from a set of premises  $A$ , is true in **all interpretations that satisfy  $A$**
  - and conversely that every statement entailed by  $A$ -interpretations 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** 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 (obviously).

## 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

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

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 Domains, ranges and open worlds

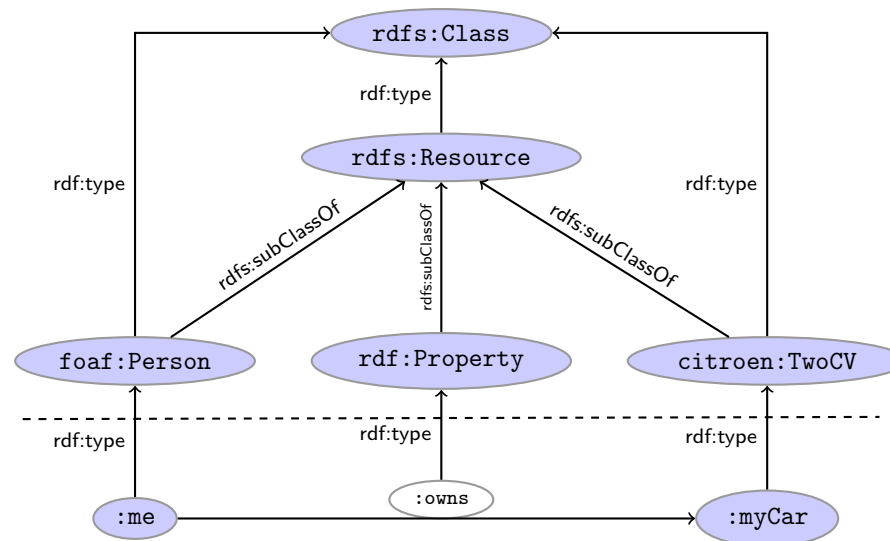
## RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  - `rdfs`  $\equiv$  `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*
- (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`: Concept 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
$A \text{ rdf:type rdfs:Class}$	$A$ is a set of resources
$x \text{ rdf:type } A$	$x \in A$
$A \text{ rdfs:subClassOf } B$	$A \subseteq B$

## 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:

- “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...”

## 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**

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

- **Transitivity of sub-class relation**

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

## Example

**RDFS/RDF knowledge base:**

`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 .`

`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:Mammal .` (rdfs11)

`ex:Mammal rdfs:subClassOf ex:Mammal .` (rdfs10)

## 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**

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

- **Transitivity of sub-class relation**

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

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

## A typical taxonomy

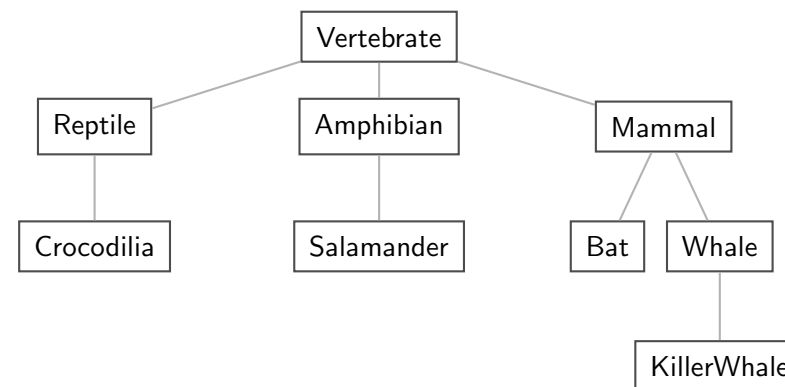


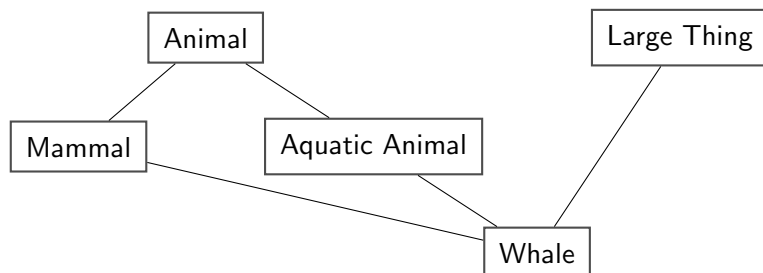
Figure: A typical taxonomy

## Multiple Inheritance

- A set is a subset of many other sets:

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

- 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
$r \text{ rdf:type } \text{rdf:Property}$	$r$ 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

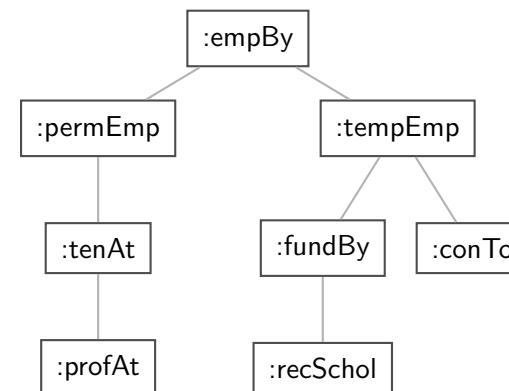


Figure: A hierarchy of employment relations

- 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:

```
: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 :

#### Temporary employees

```
SELECT ?emp WHERE {?emp :tempEmp _:x .}
→ Audun, Martin, Trond
```

#### Permanent employees

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

#### All employees

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

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

- Typing second coordinates:

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

## 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.



## 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  $x R \dots$ :

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

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

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

- Example:

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

## 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 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 :Petrenko .
```

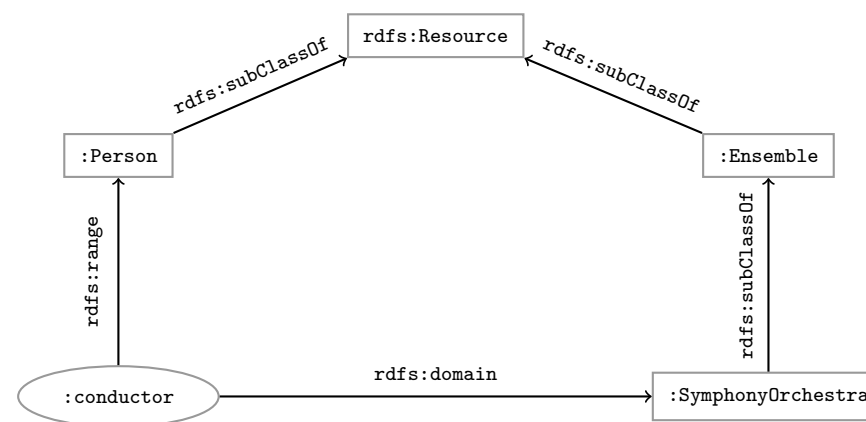
we may infer;

```
: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 .
:Martin :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 .
:Martin :conTo :OLF .
:Martin 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!

## Outline

- 1 Inference rules
- 2 RDFS Basics
- 3 Domains, ranges and open worlds

## 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 .`
  - `: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.
  - in a standard relational database,
  - it would follow that `:Boston` is **not** an `:Orchestra`,
  - which contradicts the rule `rdfs7`:

```

:isRecordedBy rdfs:range :Orchestra .           :Turangalila :isRecordedBy :Boston .
-----
:Boston rdf:type :Orchestra .                    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

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,
  - 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.

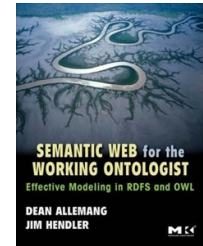
Therefore,

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

## 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.*

