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

Model-theoretic semantics, a quick recap

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

Model-theoretic semantics, a quick recap

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

- we specified in a mathematically precise way

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,

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, 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

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

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

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,

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

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,

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.

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

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,

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,

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,

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.

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,

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

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 reasoning

We therefore need means to decide entailment syntactically:

- Syntactic methods operate only on the form of a statement, that is

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,

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

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.

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

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

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:

Soundness and completeness

Semantics and calculus are typically made to work like chopsticks:

- One proves that,

Soundness and completeness

Semantics and calculus are typically made to work like chopsticks:

- One proves that,
  1. every conclusion derivable in the calculus from a set of premises A, is true in all models that satisfy A

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.

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

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

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

Inference rules

A calculus is usually formulated in terms of

Inference rules

A calculus is usually formulated in terms of

- a set of axioms that represent unquestioned postulates,

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.

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:

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:

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

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

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.

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,

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),

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

Describing models

Given soundness and completeness of a calculus wrt a semantics:

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

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 $P_1, \ldots, P_n$

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,

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:

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.

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

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

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:

RDFS axiomatic triples (excerpt)

RDFS axiomatics

- Only resources have types:

```
rdf:type rdfs:domain rdfs:Resource .
```

RDFS axiomatic triples (excerpt)

RDFS axiomatics

- Only resources have types:

```
rdf:type rdfs:domain rdfs:Resource .
```

- Domains apply only to properties:

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 .
```

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 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 .
```

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 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 .
```

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 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

Inference in the concrete

In a Semantic Web context, inference always means,

Inference in the concrete

In a Semantic Web context, inference always means,

- adding triples,

Inference in the concrete

In a Semantic Web context, inference always means,

- adding triples,

More specifically it means,

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),

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.

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

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

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

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;

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"

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

Studying RDFS through inference rules

In this lecture we shall

Studying RDFS through inference rules

In this lecture we shall

- study RDFS by example,

Studying RDFS through inference rules

In this lecture we shall

- study RDFS by example,
- by switching from the semantic point of view,

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.

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

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

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

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.

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

RDFS incompleteness

The RDFS incompleteness is manifest in certain cases of

- domain and range reasoning.

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

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,

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.

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

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,

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).

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 in a nutshell

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

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,

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,

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 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,

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 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 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:

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 ..."

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 ..."

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:**

Concepts in the RDFS datamodel

Corresponding vocabulary items

- **RDFS classes:**
  - `rdfs:Resource`: The class of resources, everything.

Concepts in the RDFS datamodel

Corresponding vocabulary items

- **RDFS classes:**
  - `rdfs:Resource`: The class of resources, everything.
  - `rdfs:Class`: The class of classes.

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`)

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**

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.

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.

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.

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:

Talking about sets and relations

Thus RDFS is about sets and relations:

- Nodes are grouped into `rdfs:Classes`.

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.

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`.

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`

Talking about sets and relations

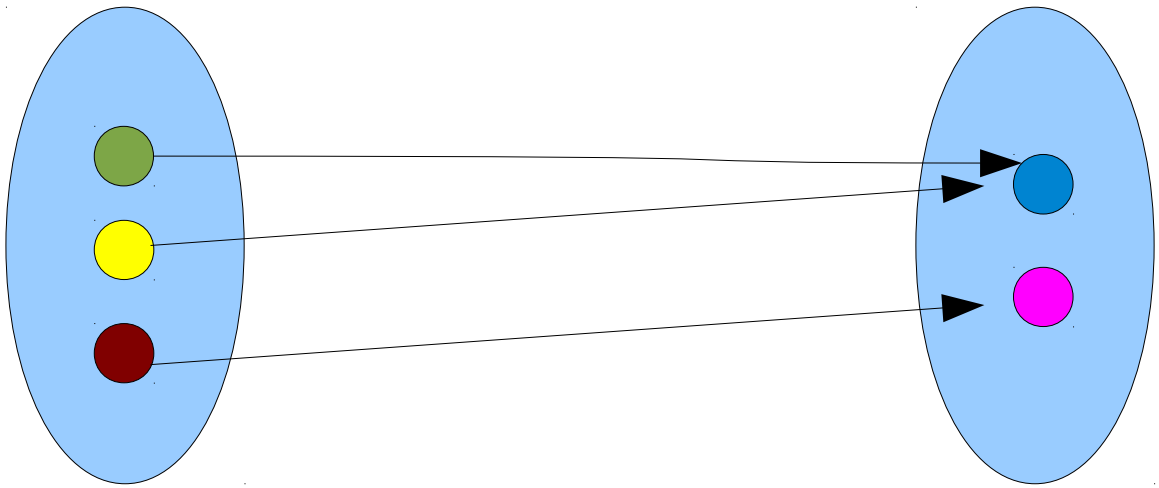
Thus RDFS is about sets and relations:

- 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`

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,

contd.

Finally,

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

contd.

Finally,

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

contd.

Finally,

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

Stated plainly RDFS is thus

contd.

Finally,

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

Stated plainly RDFS is thus

- a simple language for defining class and property taxonomies,

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,

contd.

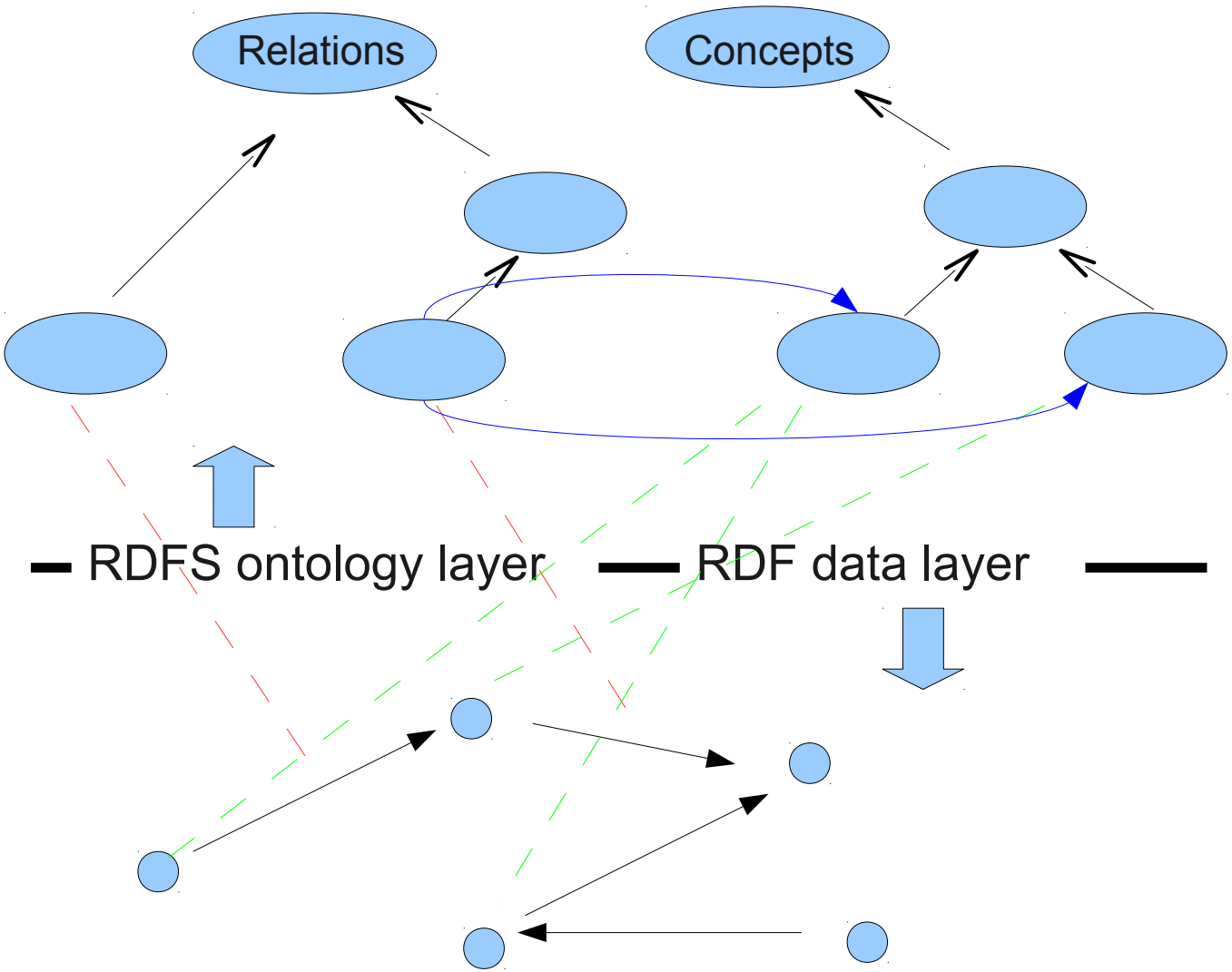
Finally,

- 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

Things to note

This is not an entirely accurate depiction, since

- there is no clear distinction between data and ontology in RDFS.

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:

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;

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

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,

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.

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`

First pattern: Type propagation with `rdfs:subClassOf`

The type propagation rules apply

First pattern: Type propagation with `rdfs:subClassOf`

The type propagation rules apply

- to combinations of `rdf:type`, `rdfs:subClassOf` and `rdfs:Class`,

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.

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:

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:

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:

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

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:

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:

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

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:

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:

Example

RDFS ontology:

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

Example

RDFS ontology:

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

Example

RDFS ontology:

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

ex:Mammal rdf:type rdfs:Class .

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

Example

RDFS ontology:

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

ex:Mammal rdf:type rdfs:Class .

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

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 .
```

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 .
```

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 .
```

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:

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 .
```

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 .
```

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:

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 .
```

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 .
```

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

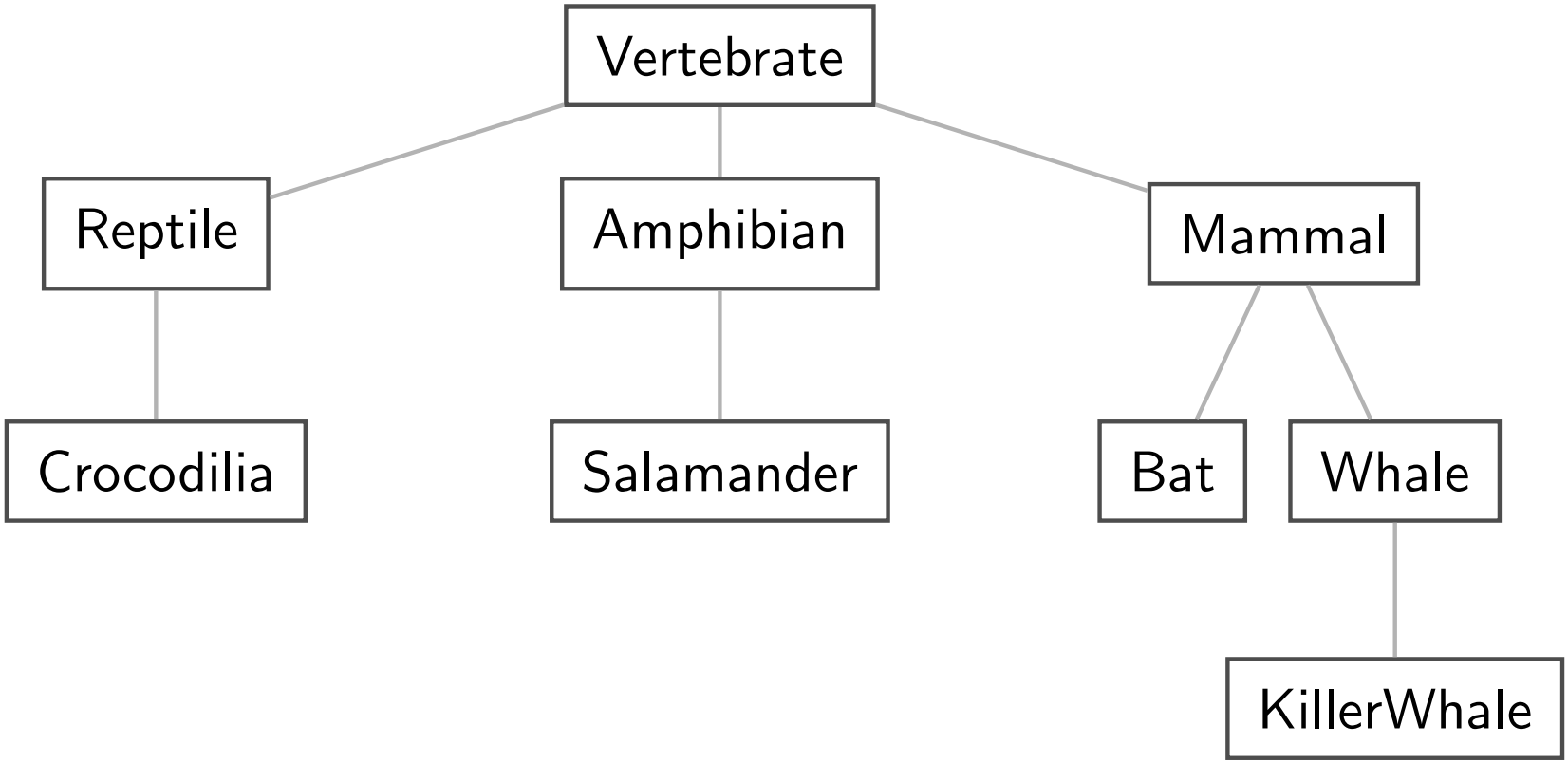


Figure: A typical taxonomy

Second: Property transfer with `rdfs:subPropertyOf`

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depend on certain combinations of

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depend on certain combinations of

- `rdfs:subPropertyOf`,

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depend on certain combinations of

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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depend on certain combinations of

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

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:

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:

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:

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

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:

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

- Reflexivity:

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:

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

- Reflexivity:

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

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:

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

- Reflexivity:

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

- Property transfer:

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:

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

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

Example I: Harmonizing terminology

Integrating data from multiple sources in general requires:

Example I: Harmonizing terminology

Integrating data from multiple sources in general requires:

- Harmonisation of the data under a common vocabulary.

Example I: Harmonizing terminology

Integrating data from multiple sources in general requires:

- Harmonisation of the data under a common vocabulary.

The aim is to

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,

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

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:

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

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`

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,

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

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

Solution

RDFS Ontology:

Solution

RDFS Ontology:

```
writer rdf:type rdf:Property .
```

Solution

RDFS Ontology:

```
writer rdf:type rdf:Property .
author rdf:type rdf:Property .
```

Solution

RDFS Ontology:

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

Solution

RDFS Ontology:

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

Solution

RDFS Ontology:

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

Effects:

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,

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.

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,

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.

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.

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

Example II: Keeping track of employees

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

Example II: Keeping track of employees

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

- for tenured positions (professors, assisting professors, lecturers),

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),

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,

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.

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.

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:

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*),

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*),

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*),

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*) ,

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

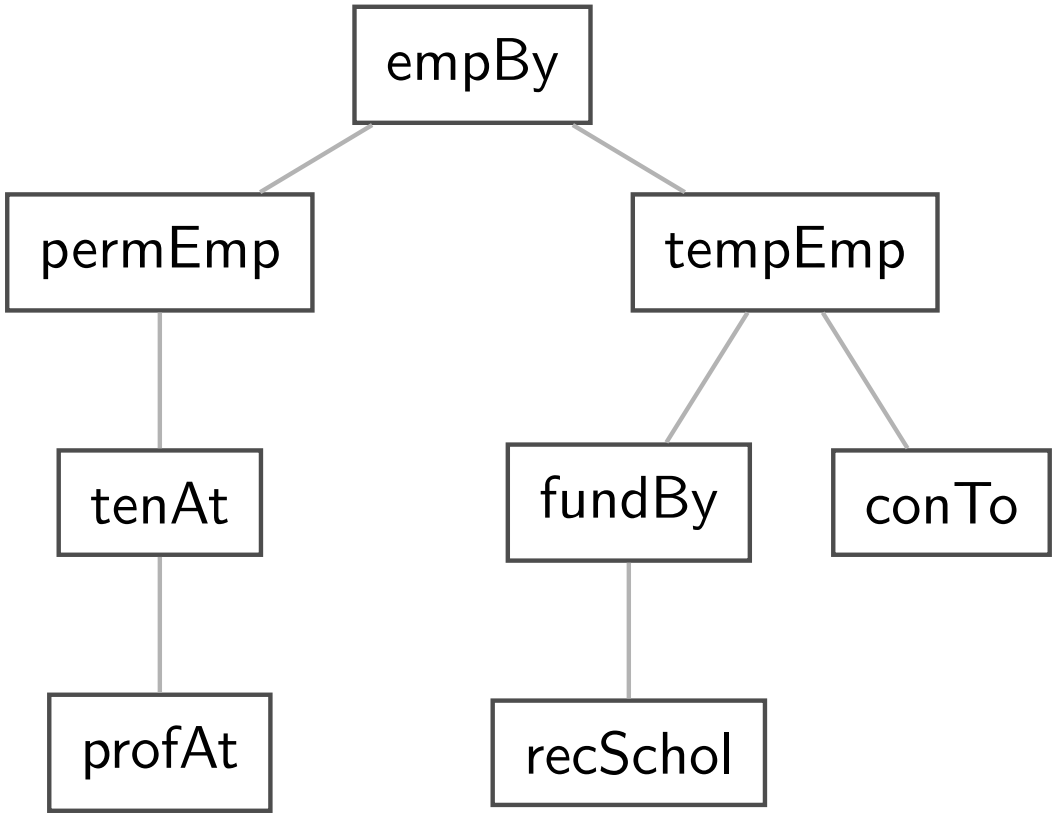


Figure: A hierarchy of employment relations

Querying the inferred model

Formalising the tree:

Querying the inferred model

Formalising the tree:

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

Querying the inferred model

Formalising the tree:

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

Querying the inferred model

Formalising the tree:

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

Querying the inferred model

Formalising the tree:

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

..... and so forth.

Querying the inferred model

Formalising the tree:

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

..... and so forth.

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:

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 .
```

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 .
```

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 .
```

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 .
```

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 :

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 .
}
```

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:

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

Third pattern: Typing data based on their use

Triggered by combinations of

Third pattern: Typing data based on their use

Triggered by combinations of

- `rdfs:range`

Third pattern: Typing data based on their use

Triggered by combinations of

- `rdfs:range`
- `rdfs:domain`

Third pattern: Typing data based on their use

Triggered by combinations of

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

Third pattern: Typing data based on their use

Triggered by combinations of

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

Rules for damain and range reasoning :

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:

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:

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

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:

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

- Typing second coordinates:

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:

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

- Typing second coordinates:

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

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:

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

- Typing second coordinates:

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

Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is used.

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,

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,

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

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`,

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`,

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 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`

Example I: Combining `domain`, `range` and `subClassOf`

Suppose we have a class tree that includes:

Example I: Combining `domain`, `range` and `subClassOf`

Suppose we have a class tree that includes:

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

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:

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 .
```

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 .
```

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

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 .
```

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;

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 .
```

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 .
```

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 .
```

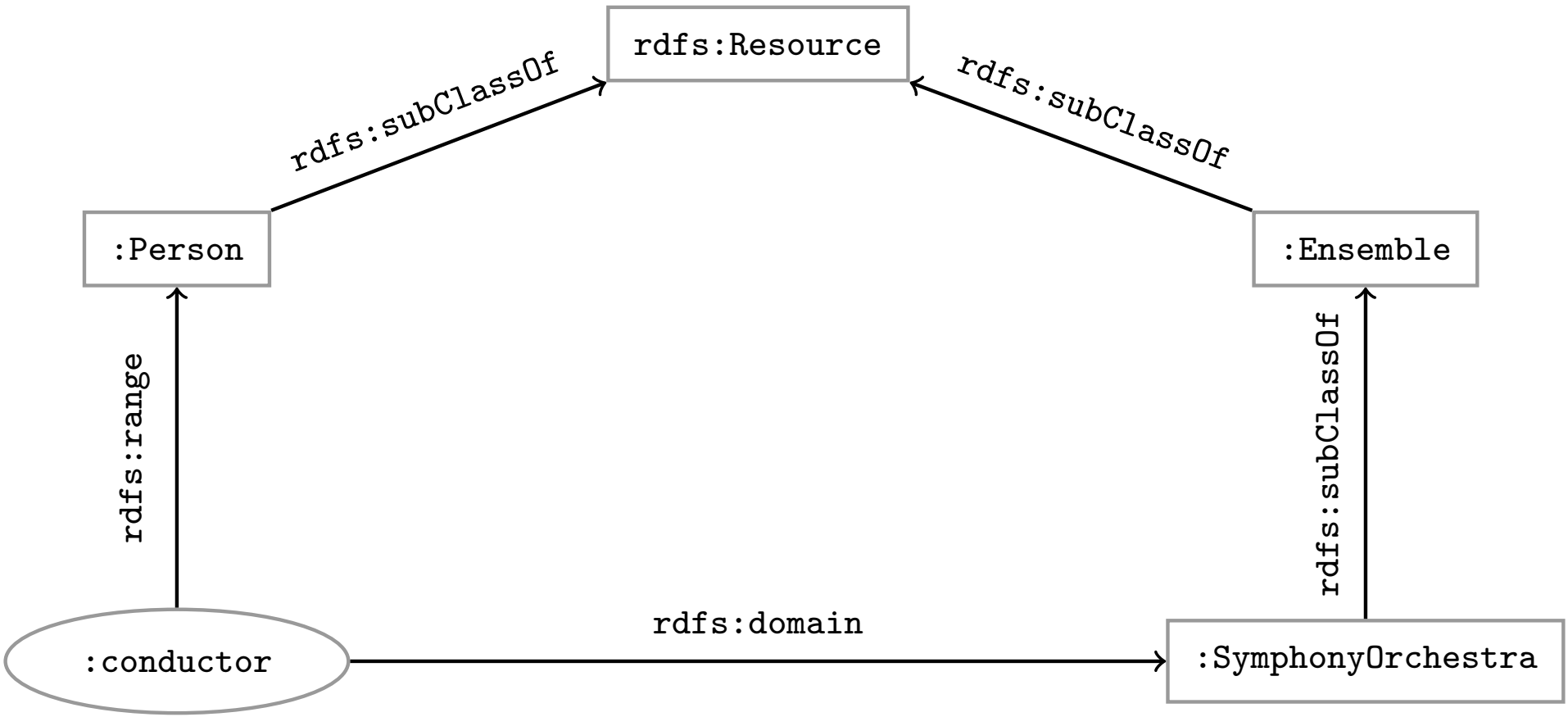
Now, if we assert

```
: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:

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 .
```

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:

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,

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`,

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`:

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 .
```

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 .
```

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,

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

Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

$$\frac{\texttt{:contractsTo rdfs:domain :Freelancer .} \qquad \texttt{:Martin :contractsTo :OLF .}}{\texttt{:Martin rdf:type :Freelancer}}\ \text{rdfs2}$$

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

Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

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

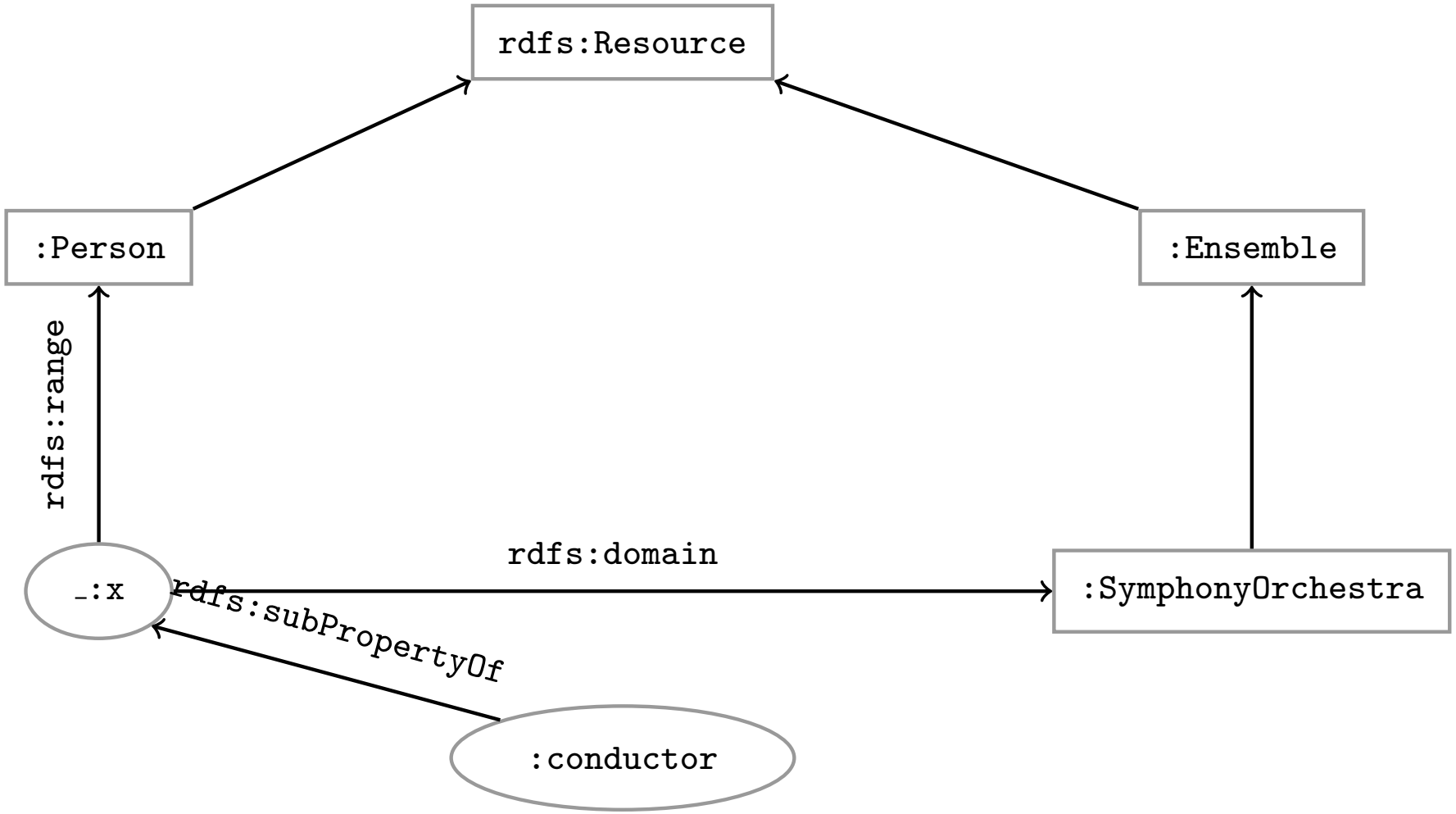
and may be used as a type in SPARQL (reasoner presupposed):

Finding the freelancers

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

A conspicuous non-pattern

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



The incompleteness of RDFS

That is:

The incompleteness of RDFS

That is:

- We make `:conductor` a subproperty of `_:x`,

The incompleteness of RDFS

That is:

- We make `:conductor` a subproperty of `_:x`,
- `_:x` is a generic relation between people and orchestras,

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.

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.

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):

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):

$$\dfrac{\dfrac{\texttt{:Oslo :vis :Abadi .} \qquad \texttt{:vis rdfs:subProp \_:x .}}{\texttt{:Oslo \_:x :Abadi .}}\,\text{rdfs7} \qquad \texttt{\_:x rdfs:domain :Person}}{\texttt{:Abadi rdfs:type :Person}}\,\text{rdfs2}$$

Contd.

Contd.

- However, we cannot use `rdfs2` and `rdfs7` in this way,

Contd.

- However, we cannot use `rdfs2` and `rdfs7` in this way,
- since it requires putting a blank in predicate position,

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.

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.

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,

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,

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,

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

Assessing the situation

RDFS reasoners usually implement only the standardised incomplete rules, so

Assessing the situation

RDFS reasoners usually implement only the standardised incomplete rules, so

- they do not guarantee complete reasoning.

Assessing the situation

RDFS reasoners usually implement only the standardised incomplete rules, so

- they do not guarantee complete reasoning.

Better therefore;

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,

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.

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

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,

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,

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.

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.

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.

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;

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 .
```

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 .
```

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`:

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.

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,

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`,

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`:

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`:

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;

Contd.

Instead;

- RDFS infers a new triple.

Contd.

Instead;

- RDFS infers a new triple.
- More specifically it adds `:Boston rdf:type :Orchestra .`

Contd.

Instead;

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

Contd.

Instead;

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

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:

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`",

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."

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

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:

Ramifications

This fact has two important consequences:

1. RDFS is useless for validation,

Ramifications

This fact has two important consequences:

1. RDFS is useless for validation,
   - ... understood as sorting conformant from non-conformant documents,

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,

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,

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,

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,

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,

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.

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.

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

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

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 .,
     ```

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 .
```

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.

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).

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,

Expressive limitations of RDFS

Hence,

- RDFS cannot express inconsistencies,

Expressive limitations of RDFS

Hence,

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

Expressive limitations of RDFS

Hence,

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

Therefore,

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.

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.

Expressive limitations of RDFS

Hence,

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

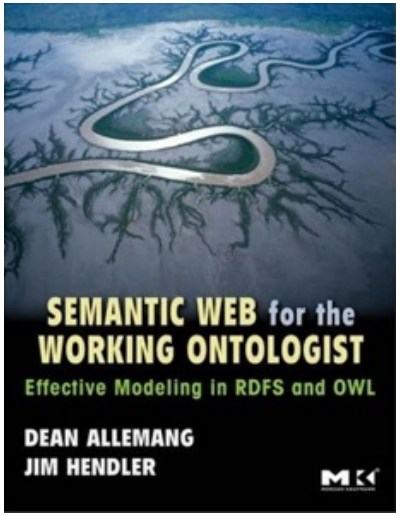
Therefore,

- 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.



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.