IN3060/4060 – Semantic Technologies – Spring 2021

Lecture 6: Introduction to Reasoning with RDF

Jieying Chen

19th February 2021

Department of Informatics

University of Oslo

Mandatory exercises

- Oblig 4 published after this lecture.

Mandatory exercises

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

Today's Plan

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

Outline

Model-theoretic semantics, a quick recap

We introduced *interpretations*:

- Idea: put all letters that are "true" into a set.

Model-theoretic semantics, a quick recap

We introduced *interpretations*:

- Idea: put all letters that are "true" into a set.
- Define: An *interpretation* $\mathcal{I}$ is a set of letters.

Model-theoretic semantics, a quick recap

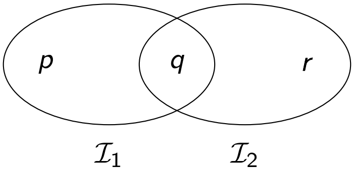
We introduced *interpretations*:

- Idea: put all letters that are "true" into a set.
- Define: An *interpretation* $\mathcal{I}$ is a set of letters.
- Letter p is true in interpretation $\mathcal{I}$ if $p \in \mathcal{I}$.

Model-theoretic semantics, a quick recap

We introduced *interpretations*:

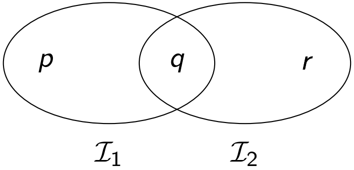
- Idea: put all letters that are "true" into a set.
- Define: An *interpretation* $\mathcal{I}$ is a set of letters.
- Letter p is true in interpretation $\mathcal{I}$ if $p \in \mathcal{I}$.
- E.g., in $\mathcal{I}_1 = \{p, q\}$, p is true, but r is false.



Model-theoretic semantics, a quick recap

We introduced *interpretations*:

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



Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

- when a formula is *true* in an interpretation: $\mathcal{I} \models A$

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics is well-suited for

- studying the behaviour of a logic, since

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics is well-suited for

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

Model-theoretic semantics, a quick recap, contd.

We specified in a mathematically precise way

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

Model-theoretic semantics is well-suited for

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

Implementational disadvantages of model semantics

Model-theoretic semantics yields an 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*,
- 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 interpretations* 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 interpretations* would have to be considered.
  - But as there can be *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 interpretations* would have to be considered.
  - But as there can be *infinitely many such interpretations*,
  - and an algorithm should terminate in *finite* time

Implementational disadvantages of model semantics

Model-theoretic semantics yields an unambigous notion of entailment,

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

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

Syntactic 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, *computation*.

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

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

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

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

Interpretations still figure as the theoretical backdrop, as one typically

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

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

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

Interpretations still figure as the theoretical backdrop, as one typically

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

Syntactic reasoning

We therefore need means to decide entailment *syntactically*:

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

Interpretations still figure as the theoretical backdrop, as one typically

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

Syntactic reasoning easier to understand and use than model semantics

- we will show that first.

Inference rules

A calculus is usually formulated in terms of

Inference rules

A calculus is usually formulated in terms of

- a set of *axioms* which are tautologies,

Inference rules

A calculus is usually formulated in terms of

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

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

- the P_i are *premises*

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

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

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

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

An inference rule may have,

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

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

An inference rule may have,

- any number of premises (typically one or two),

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

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

An inference rule may have,

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

Inference rules

A calculus is usually formulated in terms of

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

The general form of an inference rule is:

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

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

An inference rule may have,

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

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

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

- One proves that,

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

- One proves that,
  1. every conclusion P derivable in the calculus from a set of premises Γ, is true in *all interpretations that satisfy* Γ. ($\Gamma \vdash P \Rightarrow \Gamma \models P$)

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

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

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

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

We say that the calculus is

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

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

We say that the calculus is

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

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

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

We say that the calculus is

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

Soundness and completeness

Semantics and calculus are typically made to work in pairs:

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

We say that the calculus is

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

Inference rules in propositional logic

(Part of) Natural deduction calclulus for propositional logic:

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

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

Inference for RDF

Inference for RDF

In a Semantic Web context, inference always means,

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF graph,

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF graph,
- on the basis of the triples *already in it*.

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF graph,
- on the basis of the triples *already in it*.

From this point of view a rule

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF graph,
- on the basis of the triples *already in it*.

From this point of view a rule

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

may be read as an instruction;

Inference for RDF

In a Semantic Web context, inference always means,

- *adding triples*.

More specifically it means,

- adding *new triples* to an RDF graph,
- on the basis of the triples *already in it*.

From this point of view a rule

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

may be read as an instruction;

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

Outline

RDF Schema

- RDF Schema is a vocabulary defined by W3C.

RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`

RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`
- Originally thought of as a "schema language" like XML Schema.

RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`
- Originally thought of as a "schema language" like XML Schema.
- Actually it isn't – doesn't describe "valid" RDF graphs.

RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`
- Originally thought of as a "schema language" like XML Schema.
- Actually it isn't – doesn't describe "valid" RDF graphs.
- Comes with some inference rules

RDF Schema

- RDF Schema is a vocabulary defined by W3C.
- Namespace:
  `rdfs: http://www.w3.org/2000/01/rdf-schema#`
- Originally thought of as a "schema language" like XML Schema.
- Actually it isn't – doesn't describe "valid" RDF graphs.
- Comes with some inference rules
  - Allows to derive new triples mechanically.

RDF Schema

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

RDF Schema

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

RDF Schema concepts

- RDFS adds the concept of “classes” which are like *types* or *sets* of resources.

RDF Schema concepts

- RDFS adds the concept of “classes” which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.

RDF Schema concepts

- RDFS adds the concept of "classes" which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:

RDF Schema concepts

- RDFS adds the concept of "classes" which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
    - `rdfs:Resource`: The class of resources, everything.

RDF Schema concepts

- RDFS adds the concept of "classes" which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
    - `rdfs:Resource`: The class of resources, everything.
    - `rdfs:Class`: The class of classes.

RDF Schema concepts

- RDFS adds the concept of "classes" which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
  - `rdfs:Resource`: The class of resources, everything.
  - `rdfs:Class`: The class of classes.
  - `rdf:Property`: The class of properties (from `rdf`).

RDF Schema concepts

- RDFS adds the concept of “classes” which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
  - `rdfs:Resource`: The class of resources, everything.
  - `rdfs:Class`: The class of classes.
  - `rdf:Property`: The class of properties (from `rdf`).
- Defined properties:

RDF Schema concepts

- RDFS adds the concept of "classes" which are like *types* or *sets* of resources.
- The RDFS vocabulary allows statements about classes.
- Defined resources:
  - `rdfs:Resource`: The class of resources, everything.
  - `rdfs:Class`: The class of classes.
  - `rdf:Property`: The class of properties (from `rdf`).
- Defined properties:
  - `rdf:type`: relate resources to classes they are members of.

RDF Schema concepts

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

RDF Schema concepts

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

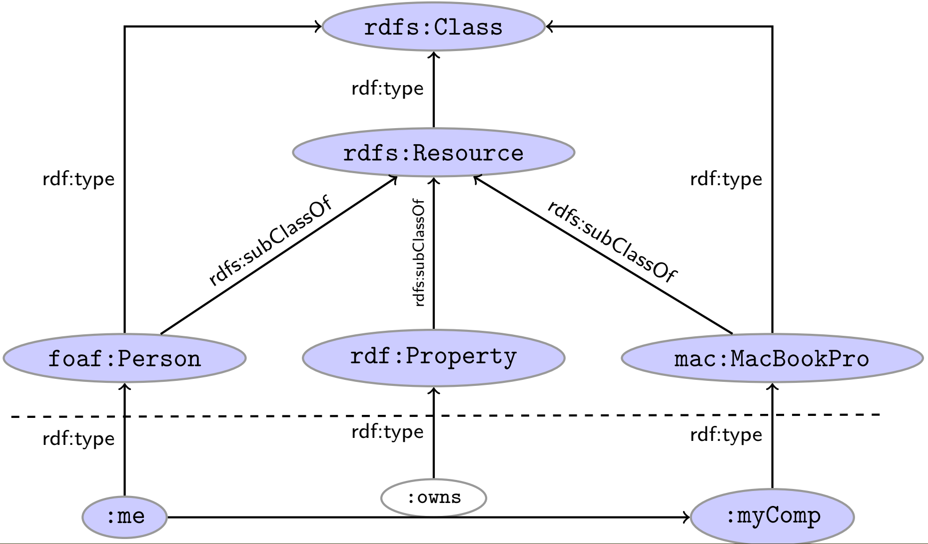
RDF Schema concepts

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

RDF Schema concepts

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

Example


rdfs:Class
rdf:type
rdfs:Resource
rdf:type
rdfs:subClassOf
rdfs:subClassOf
rdfs:subClassOf
rdf:type
foaf:Person
rdf:Property
mac:MacBookPro
rdf:type
rdf:type
rdf:type
:owns
:me
:myComp

Intuition: Classes as Sets

- We can think of an `rdfs:Class` as denoting a *set* of Resources.

Intuition: Classes as Sets

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

Intuition: Classes as Sets

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

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

Simple Entailment Rules

- Entailment with blank nodes and literals

Simple Entailment Rules

- Entailment with blank nodes and literals
- Without RDFS and RDF axioms

Simple Entailment Rules

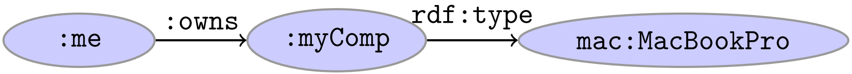
- Entailment with blank nodes and literals
- Without RDFS and RDF axioms
- $\dfrac{\texttt{A R B .}}{\texttt{A R \_:x .}}\,\text{se1} \qquad \dfrac{\texttt{A R B .}}{\texttt{\_:x R B .}}\,\text{se2}$

Simple Entailment Rules

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

Simple Entailment Example

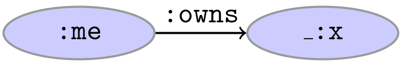
- Let's create the RDF-graph with the two triples:
  1. `:me :owns :myComp .`
  2. `:myComp rdf:type mac:MacBookPro .`



Simple Entailment Example

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

Using **se1** on triple 1, it entails: `:me :owns _:x`
`_:x` is not used previously

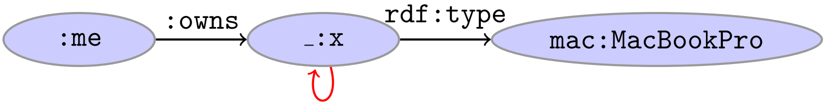


Simple Entailment Example

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

Using **se1** on triple 1, it entails: `:me :owns _:x`
`_:x` is not used previously

Using **se2** on triple 2 it entails: `_:x rdf:type mac:MacBookPro`
`_:x` refers to the same URI



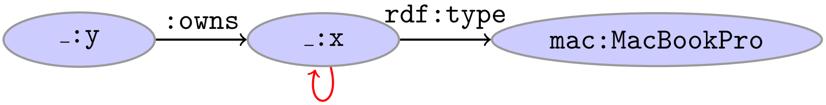
Simple Entailment Example

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

Using **se1** on triple 1, it entails: `:me :owns _:x`
`_:x` is not used previously

Using **se2** on triple 2 it entails: `_:x rdf:type mac:MacBookPro`
`_:x` refers to the same URI

Using **se2** the inferred triple `:me :owns _:x`
entails `_:y :owns _:x` where `_:y` refers to the new URI



Simple Entailment Example

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

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

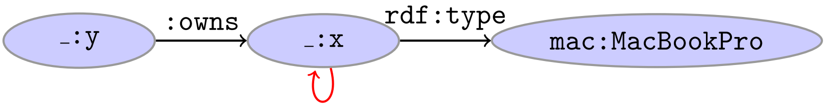
`_:x` is not used previously

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

`_:x` refers to the same URI

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

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



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

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so. . ."

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so..."

II. *Property inheritance:*

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so..."

II. *Property inheritance:*
   - "Steve *lectures at* Ifi, and anyone who does so is *employed by* Ifi, so..."

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so. . ."

II. *Property inheritance:*
   - "Steve *lectures at* Ifi, and anyone who does so is *employed by* Ifi, so. . ."

III. *Domain and range reasoning:*

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so. . . "

II. *Property inheritance:*
   - "Steve *lectures at* Ifi, and anyone who does so is *employed by* Ifi, so. . . "

III. *Domain and range reasoning:*
   - "Everything someone *has written* is a *document*. Alan *has written* 'Computing Machinery and Intelligence', therefore. . . "

RDFS reasoning

RDFS supports three principal kinds of *reasoning pattern*:

I. *Type propagation:*
   - "The 2CV *is a car*, and all cars *are motorised vehicles*, so…"

II. *Property inheritance:*
   - "Steve *lectures at* Ifi, and anyone who does so is *employed by* Ifi, so…"

III. *Domain and range reasoning:*
   - "Everything someone *has written* is a *document*. Alan *has written* 'Computing Machinery and Intelligence', therefore…"
   - "All *father*s of people are *male*s. James is the *father* of Karl, therefore…"

Type propagation with rdfs:subClassOf

The type propagation rules apply

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

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 with `rdfs:subClassOf`

The type propagation rules apply

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

Type propagation rules:

- *Members of subclasses*

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

Type propagation with `rdfs:subClassOf`

The type propagation rules apply

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

Type propagation rules:

- *Members of subclasses*

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

- *Reflexivity of sub-class relation*

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

Type propagation with `rdfs:subClassOf`

The type propagation rules apply

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

Type propagation rules:

- *Members of subclasses*

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

- *Reflexivity of sub-class relation*

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

- *Transitivity of sub-class relation*

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

Set Theory Analogy

- *Members of subclasses*

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

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

Set Theory Analogy

- *Members of subclasses*

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

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

- *Reflexivity of sub-class relation*

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

$$\frac{A \text{ is a set}}{A \subseteq A}$$

Set Theory Analogy

- *Members of subclasses*

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

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

- *Reflexivity of sub-class relation*

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

- *Transitivity of sub-class relation*

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

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

Example

RDFS/RDF knowledge base:

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

Example

RDFS/RDF knowledge base:

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

Example

RDFS/RDF knowledge base:

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

Example

RDFS/RDF knowledge base:

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

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

Example

RDFS/RDF knowledge base:

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

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

Example

RDFS/RDF knowledge base:

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

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

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

Example

RDFS/RDF knowledge base:

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

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

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

Example

RDFS/RDF knowledge base:

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

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

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

Inferred triples:

`ex:Keiko rdf:type ex:Mammal .` (rdfs9)

Example

RDFS/RDF knowledge base:

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

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

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

Inferred triples:

`ex:Keiko rdf:type ex:Mammal .` (rdfs9)

`ex:Keiko rdf:type ex:Vertebrate .` (rdfs9)

Example

RDFS/RDF knowledge base:

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

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

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

Inferred triples:

`ex:Keiko rdf:type ex:Mammal .` (rdfs9)

`ex:Keiko rdf:type ex:Vertebrate .` (rdfs9)

`ex:KillerWhale rdfs:subClassOf ex:Vertebrate .` (rdfs11)

Example

RDFS/RDF knowledge base:

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

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

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

Inferred triples:

`ex:Keiko rdf:type ex:Mammal .` (rdfs9)

`ex:Keiko rdf:type ex:Vertebrate .` (rdfs9)

`ex:KillerWhale rdfs:subClassOf ex:Vertebrate .` (rdfs11)

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

(... and also for the other classes)

A typical taxonomy

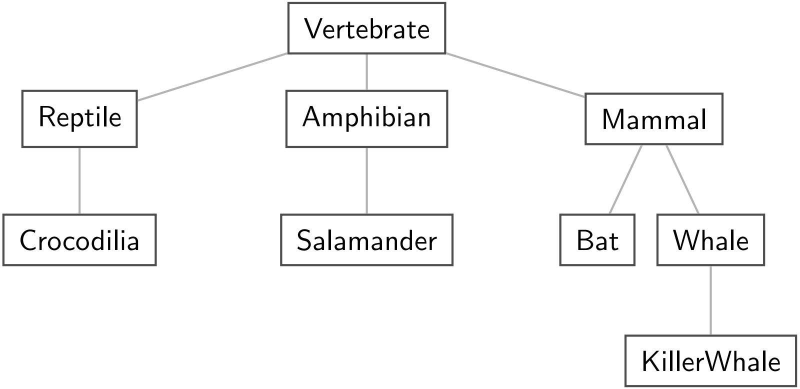


Figure: A typical taxonomy

Multiple Inheritance

- A set is a subset of many other sets:

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

Multiple Inheritance

- A set is a subset of many other sets:

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

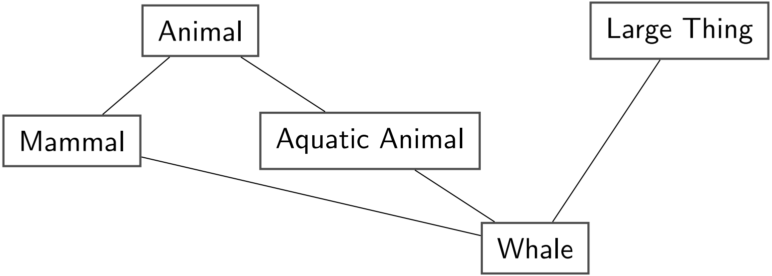
- Similarly, a class is usually a subclass of many other classes.

Multiple Inheritance

- A set is a subset of many other sets:

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

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

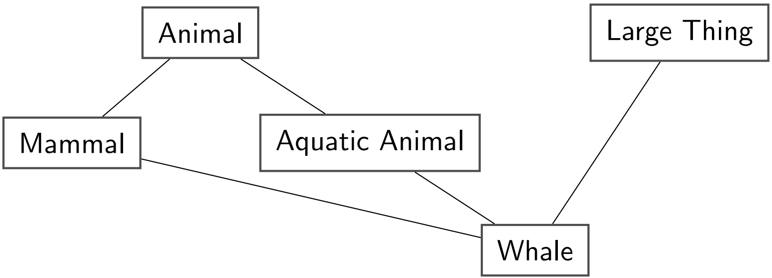


Multiple Inheritance

- A set is a subset of many other sets:

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

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



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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

- `rdfs:subPropertyOf`,

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

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

- *Reflexivity:*

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

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

- *Reflexivity:*

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

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

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

- *Reflexivity:*

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

- *Property transfer:*

Second: Property transfer with `rdfs:subPropertyOf`

Reasoning with properties depends on certain combinations of

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

Rules for property reasoning:

- *Transitivity:*

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

- *Reflexivity:*

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

- *Property transfer:*

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

Intuition: Properties as Relations

- If an `rdfs:Class` is like a set of resources. . .

Intuition: Properties as Relations

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

Intuition: Properties as Relations

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

Intuition: Properties as Relations

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

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

Intuition: Properties as Relations

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

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

- Rules:

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

Example I: Harmonizing terminology

Integrating data from multiple sources in general requires:

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.

Solution

From Ontology:

Solution

From Ontology:

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

Solution

From Ontology:

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

And Facts:

Solution

From Ontology:

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

And Facts:

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

Infer:

Solution

From Ontology:

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

And Facts:

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

Infer:

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

Consequences

- Any individual for which `:author` or `:writer` is defined,

Consequences

- Any individual for which `:author` or `:writer` is defined,
- will have the same value for the `dcterms:creator` property.

Consequences

- Any individual for which `:author` or `:writer` is defined,
- will have the same value for the `dcterms:creator` property.
- The work of integrating the data is thus done by the reasoning engine,

Consequences

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

Consequences

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

Example II: Keeping track of employees

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

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

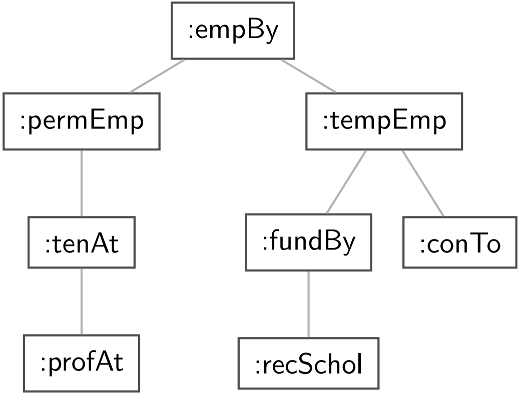


Figure: A hierarchy of employment relations

Organising the properties

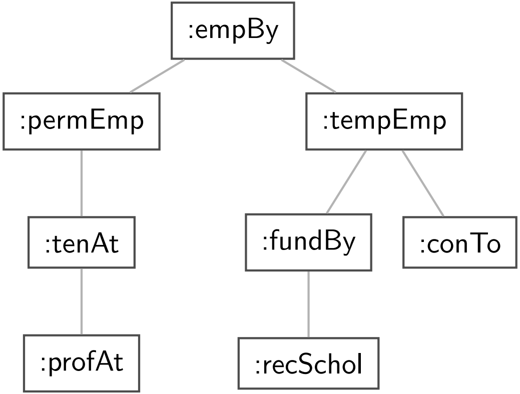


Figure: A hierarchy of employment relations

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

Querying the inferred model

Formalising the tree:

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

..... and so forth.

Querying the inferred model

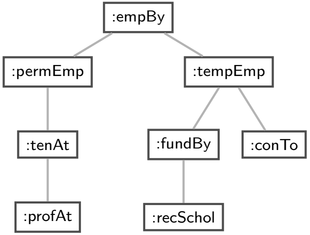
Formalising the tree:

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

..... and so forth.

Given a data set such as:

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



cont.

We may now query on different levels of abstraction :

Temporary employees

```
SELECT ?emp WHERE {?emp :tempEmp _:x .}
```

→ *Ole, Steve, Trond*

cont.

We may now query on different levels of abstraction :

Temporary employees

```
SELECT ?emp WHERE {?emp :tempEmp _:x .}
```

→ *Ole, Steve, Trond*

Permanent employees

```
SELECT ?emp WHERE {?emp :permEmp _:x .}
```

→ *Martin, Jenny*

cont.

We may now query on different levels of abstraction :

Temporary employees

```
SELECT ?emp WHERE {?emp :tempEmp _:x .}
```

→ *Ole, Steve, Trond*

Permanent employees

```
SELECT ?emp WHERE {?emp :permEmp _:x .}
```

→ *Martin, Jenny*

All employees

```
SELECT ?emp WHERE {?emp :empBy _:x .}
```

→ *Martin, Jenny, Ole, Steve, Trond*

Third pattern: Typing data based on their use

Triggered by combinations of

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 domain and range reasoning :

Third pattern: Typing data based on their use

Triggered by combinations of

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

Rules for domain and range reasoning :

- *Typing first coordinates:*

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

Third pattern: Typing data based on their use

Triggered by combinations of

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

Rules for domain and range reasoning :

- *Typing first coordinates:*

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

- *Typing second coordinates:*

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

Domain and range contd.

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

Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is *used*.
- `rdfs:domain` types the *possible subjects* of these triples,

Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is *used*.
- `rdfs:domain` types the *possible subjects* of these triples,
- whereas `rdfs:range` types the *possible objects*,

Domain and range contd.

- `rdfs:domain` and `rdfs:range` tell us how a property is *used*.
- `rdfs:domain` types the *possible subjects* of these triples,
- whereas `rdfs:range` types the *possible objects*,
- When we assert that property `p` has `domain` `C`, we are saying

Domain and range contd.

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

Domain and range contd.

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

Domain and Range of Relations

- Given a relation R from A to B ($R \subseteq A \times B$)

Domain and Range of Relations

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

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

Domain and Range of Relations

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

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

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

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

Domain and Range of Relations

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

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

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

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

- Example:

Domain and Range of Relations

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

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

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

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

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

Domain and Range of Relations

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

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

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

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

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

Domain and Range of Relations

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

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

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

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

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

Set intuitions for rdfs:domain and rdfs:range

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

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

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

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

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

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

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

- Rules:

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

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

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

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

Suppose we have a class hierarchy that includes:

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

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

Suppose we have a class hierarchy that includes:

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

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

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

Suppose we have a class hierarchy that includes:

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

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

```
:conductor rdfs:domain :SymphonyOrchestra .

:conductor rdfs:range :Person .
```

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

Suppose we have a class hierarchy that includes:

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

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

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

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

Now, if we assert

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

we may infer;

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

Suppose we have a class hierarchy that includes:

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

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

```
:conductor rdfs:domain :SymphonyOrchestra .
:conductor rdfs:range :Person .
```

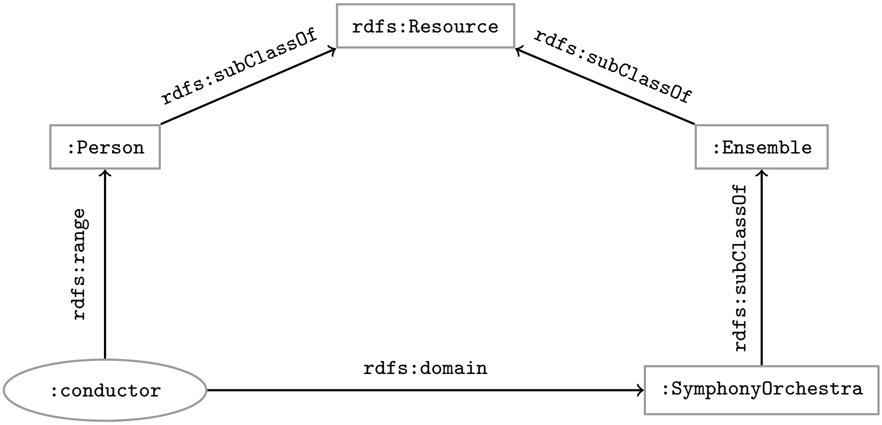
Now, if we assert

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

we may infer;

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

Conductors and ensembles



Example II: Filtering information based on use

Consider once more the dataset:

```
:Martin :profAt :UiO .

:Ole :fundBy :UiO .

:Steve :conTo :OLF .

:Trond :recSchol :BI .

:Jenny :tenAt :SSB .
```

Example II: Filtering information based on use

Consider once more the dataset:

```
:Martin :profAt :UiO .

:Ole :fundBy :UiO .

:Steve :conTo :OLF .

:Trond :recSchol :BI .

:Jenny :tenAt :SSB .
```

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

Example II: Filtering information based on use

Consider once more the dataset:

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

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

- State that only freelancers `:conTo` an organisation,

Example II: Filtering information based on use

Consider once more the dataset:

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

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

- State that only freelancers `:conTo` an organisation,
- i.e. introduce a class `:Freelancer`,

Example II: Filtering information based on use

Consider once more the dataset:

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

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

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

Example II: Filtering information based on use

Consider once more the dataset:

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

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

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

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

Example II: Filtering information based on use

Consider once more the dataset:

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

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

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

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

Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

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

Finding the freelancers

The class of freelancers is generated by the rdfs2 rule,

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

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

Finding the freelancers

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

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

```
rdfs:range rdfs:domain rdf:Property .
```

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

```
rdfs:range rdfs:domain rdf:Property .
```

- Ranges are classes:

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

  ```
  rdf:type rdfs:range rdfs:Class .
  ```

- Ranges apply only to properties:

  ```
  rdfs:range rdfs:domain rdf:Property .
  ```

- Ranges are classes:

  ```
  rdfs:range rdfs:range rdfs:Class .
  ```

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

```
rdfs:range rdfs:domain rdf:Property .
```

- Ranges are classes:

```
rdfs:range rdfs:range rdfs:Class .
```

- Only properties have subproperties:

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

```
rdfs:range rdfs:domain rdf:Property .
```

- Ranges are classes:

```
rdfs:range rdfs:range rdfs:Class .
```

- Only properties have subproperties:

```
rdfs:subPropertyOf rdfs:domain rdf:Property .
```

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

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

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

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

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

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

- types are classes:

```
rdf:type rdfs:range rdfs:Class .
```

- Ranges apply only to properties:

```
rdfs:range rdfs:domain rdf:Property .
```

- Ranges are classes:

```
rdfs:range rdfs:range rdfs:Class .
```

- Only properties have subproperties:

```
rdfs:subPropertyOf rdfs:domain rdf:Property .
```

- Only classes have subclasses:

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

RDFS axiomatic triples (excerpt)

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

- Only resources have types:

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

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

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

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

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

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

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`
  - `:conductor rdf:type rdfs:Resource`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`
  - `:conductor rdf:type rdfs:Resource`
  - `rdf:Property rdf:type rdfs:Class`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`
  - `:conductor rdf:type rdfs:Resource`
  - `rdf:Property rdf:type rdfs:Class`
  - `:Person rdfs:type rdfs:Resource`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`
  - `:conductor rdf:type rdfs:Resource`
  - `rdf:Property rdf:type rdfs:Class`
  - `:Person rdfs:type rdfs:Resource`
  - `rdfs:Class rdfs:type rdfs:Class`

Using the Axiomatic Triples

- From the statement
  `:conductor rdfs:range :Person`
- We can derive:
  - `:conductor rdf:type rdf:Property`
  - `:Person rdf:type rdfs:Class`
  - `:conductor rdf:type rdfs:Resource`
  - `rdf:Property rdf:type rdfs:Class`
  - `:Person rdfs:type rdfs:Resource`
  - `rdfs:Class rdfs:type rdfs:Class`
  - ...

Using the Axiomatic Triples

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

Writing proofs

When writing proofs, we:

- write one triple per line,

Writing proofs

When writing proofs, we:

- write one triple per line,
- enumerate the lines,

Writing proofs

When writing proofs, we:

- write one triple per line,
- enumerate the lines,
- write the rule name along with the line numbers corresponding to the assumptions,

Writing proofs

When writing proofs, we:

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

Writing proofs

When writing proofs, we:

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

Writing proofs

When writing proofs, we:

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

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

Writing proofs

When writing proofs, we:

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

```
:SymphonyOrchestra rdfs:subClassOf :Ensemble .
:conductor rdfs:domain :SymphonyOrchestra .
```

Writing proofs

When writing proofs, we:

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

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

Writing proofs

When writing proofs, we:

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

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

Writing proofs

When writing proofs, we:

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

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

- We write:

Writing proofs

When writing proofs, we:

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

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

- We write:
  1. `:OsloPhilharmonic :conductor :Petrenko .` – P

Writing proofs

When writing proofs, we:

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

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

- We write:
  1. `:OsloPhilharmonic :conductor :Petrenko .` – P
  2. `:conductor rdfs:domain :SymphonyOrchestra .` – P

Writing proofs

When writing proofs, we:

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

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

- We write:
  1. `:OsloPhilharmonic :conductor :Petrenko .` – P
  2. `:conductor rdfs:domain :SymphonyOrchestra .` – P
  3. `:OsloPhilharmonic rdf:type :SymphonyOrchestra .` – rdfs3, 1, 2

Writing proofs

When writing proofs, we:

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

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

- We write:
  1. `:OsloPhilharmonic :conductor :Petrenko .` – P
  2. `:conductor rdfs:domain :SymphonyOrchestra .` – P
  3. `:OsloPhilharmonic rdf:type :SymphonyOrchestra .` – rdfs3, 1, 2
  4. `:SymphonyOrchestra rdfs:subClassOf :Ensemble .` – P

Writing proofs

When writing proofs, we:

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

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

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

Outline

Forward chaining vs. backward chaining

Forward chaining:

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Backward chaining:

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Backward chaining:

- reasoning from conclusions to premises

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Backward chaining:

- reasoning from conclusions to premises
- '...what needs to be true for this conclusion to hold?'

Forward chaining vs. backward chaining

Forward chaining:

- reasoning from premises to conclusions of rules
- adds facts corresponding to the conclusions of rules
- entailed facts are stored and reused
- reasoning is up front

Backward chaining:

- reasoning from conclusions to premises
- '. . . what needs to be true for this conclusion to hold?'
- reasoning is on-demand

Forward chaining inference

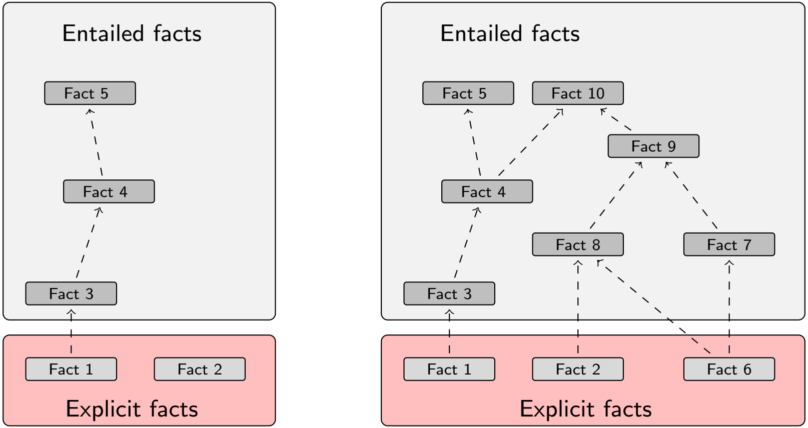


Figure: When a fact is added, all entailments are computed and stored.

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,
- and small enough to store efficiently.

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,
- and small enough to store efficiently.

Benefits:

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,
- and small enough to store efficiently.

Benefits:

- forward chaining optimizes retrieval

Benefits of forward chaining

Precomputing and storing answers is suitable for data which is:

- frequently accessed,
- expensive to compute,
- relatively static,
- and small enough to store efficiently.

Benefits:

- forward chaining optimizes retrieval
- no additional inference is necessary at query time

Forward chaining and truth-maintenance

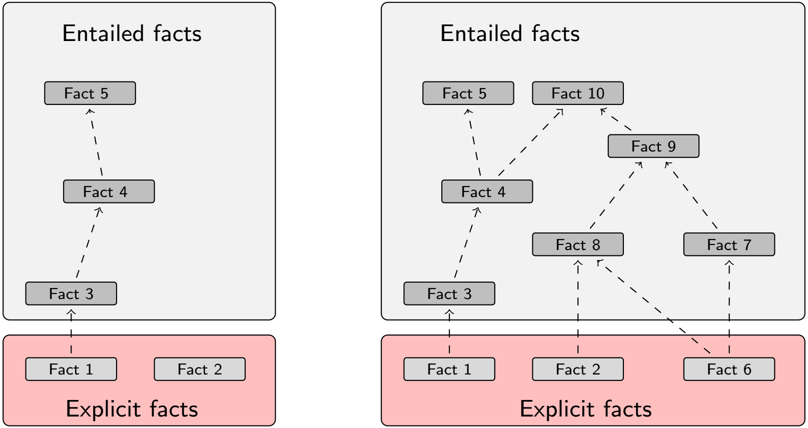


Figure: When a fact is added, all entailments are computed and stored.

Forward chaining and truth-maintenance

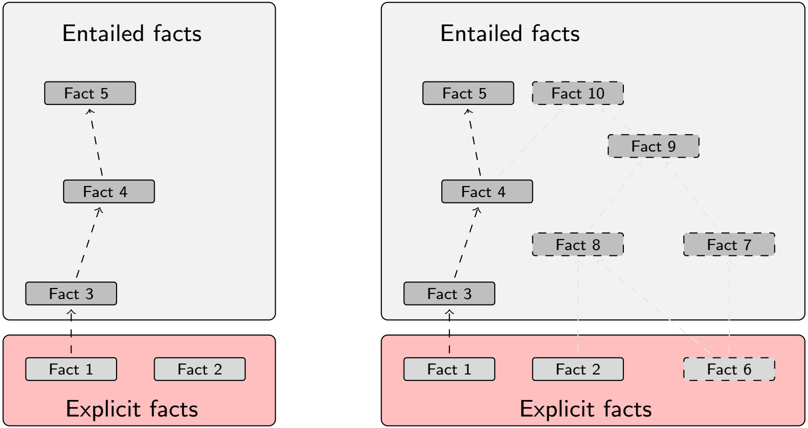


Figure: When a fact is removed, everything that comes with it must go too.

Drawbacks of forward chaining

Drawbacks:

Drawbacks of forward chaining

Drawbacks:

- increases storage size

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic
- truth maintenance usually not implemented in RDF stores

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic
- truth maintenance usually not implemented in RDF stores
- problematic for distributed and/or dynamic systems

Drawbacks of forward chaining

Drawbacks:

- increases storage size
- increases the overhead of insertion
- removal is highly problematic
- truth maintenance usually not implemented in RDF stores
- problematic for distributed and/or dynamic systems
  - rules could apply to premisses on different disks, etc.

Backward chaining inference

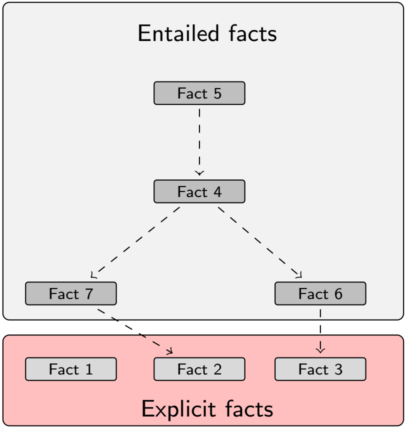


Figure: Backward chaining uses rules to expand queries.

Backward chaining: Example

RDFS/RDF knowledge base:

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

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

Backward chaining: Example

RDFS/RDF knowledge base:

```
ex:Mammal rdfs:subClassOf ex:Vertebrate .
ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Lion rdfs:subClassOf ex:Mammal .
ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

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

Query:

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

Inferred triples:

Backward chaining: Example

RDFS/RDF knowledge base:

```
ex:Mammal rdfs:subClassOf ex:Vertebrate .
ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Lion rdfs:subClassOf ex:Mammal .
ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

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

Query:

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

Inferred triples:

```
?x rdf:type ex:Vertabrate .
```

Backward chaining: Example

RDFS/RDF knowledge base:

```
ex:Mammal rdfs:subClassOf ex:Vertebrate .
ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Lion rdfs:subClassOf ex:Mammal .
ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

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

Query:

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

Inferred triples:

`?x rdf:type ex:Vertabrate .`

`?x rdf:type ex:Mammal .` (rdfs9)

Backward chaining: Example

RDFS/RDF knowledge base:

```
ex:Mammal rdfs:subClassOf ex:Vertebrate .
ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Lion rdfs:subClassOf ex:Mammal .
ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

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

Query:

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

Inferred triples:

`?x rdf:type ex:Vertabrate .`

`?x rdf:type ex:Mammal .` (rdfs9)

`?x rdf:type ex:KillerWhale .` (rdfs9) $\Rightarrow$ `?x = ex:Keiko`

Backward chaining: Example

RDFS/RDF knowledge base:

```
ex:Mammal rdfs:subClassOf ex:Vertebrate .
ex:KillerWhale rdfs:subClassOf ex:Mammal .
ex:Lion rdfs:subClassOf ex:Mammal .
ex:Keiko rdf:type ex:KillerWhale .
ex:Simba rdf:type ex:Lion .
```

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

Query:

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

Inferred triples:

`?x rdf:type ex:Vertabrate .`

`?x rdf:type ex:Mammal .` (rdfs9)

`?x rdf:type ex:KillerWhale .` (rdfs9) $\Rightarrow$ `?x = ex:Keiko`

`?x rdf:type ex:Lion .` (rdfs9) $\Rightarrow$ `?x = ex:Simba`

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic
- no persistent storage space needed

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic
- no persistent storage space needed

Drawbacks:

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic
- no persistent storage space needed

Drawbacks:

- trades insertion overhead for access overhead

Drawbacks and benefits of backward chaining

Computing answers on demand is suitable where:

- there is little need for reuse of computed answers
- answers can be efficiently computed at runtime
- answers come from multiple dynamic sources

Benefits:

- only the relevant inferences are drawn
- truth maintenance is automatic
- no persistent storage space needed

Drawbacks:

- trades insertion overhead for access overhead
- without caching, answers must be recomputed every time

Outline

Quick facts

In Jena there is

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that. . .

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
    - built-in- and

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`
- Different reasoners implement different logics, e.g

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that. . .

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- . . . and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`
- Different reasoners implement different logics, e.g
  - Transitive reasoning,

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`
- Different reasoners implement different logics, e.g
  - Transitive reasoning,
  - RDFS,

Quick facts

In Jena there is

- a zillion ways to configure and plug-in a reasoner
- some seem rather haphazard

Imposing order at the cost of precision we may say that...

- reasoners fall into one of two categories
  - built-in- and
  - external reasoners
- ...and are combined with two kinds of model
  - models of type `InfModel`, and
  - models of type `OntModel`
- Different reasoners implement different logics, e.g
  - Transitive reasoning,
  - RDFS,
  - OWL

The road most often travelled. . .

- Convenience methods are used to construct standard reasoners or inference models

The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
```

The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:
  ```
  Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
  ```
- Get inference models with standard reasoners from `ModelFactory`:
  ```
  InfModel inf = ModelFactory.createRDFSModel(rawModel);
  ```

The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:
  `Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();`
- Get inference models with standard reasoners from `ModelFactory`:
  `InfModel inf = ModelFactory.createRDFSModel(rawModel);`
- What's the point of the long winded way?

The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
```

- Get inference models with standard reasoners from `ModelFactory`:

```
InfModel inf = ModelFactory.createRDFSModel(rawModel);
```

- What's the point of the long winded way?
  - Can ask for non-builtin provers, e.g. Pellet

The road most often travelled...

- Convenience methods are used to construct standard reasoners or inference models
- Get standard reasoners from `ReasonerRegistry`:

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
```

- Get inference models with standard reasoners from `ModelFactory`:

```
InfModel inf = ModelFactory.createRDFSModel(rawModel);
```

- What's the point of the long winded way?
  - Can ask for non-builtin provers, e.g. Pellet
  - Can configure reasoners

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;
  - `getDeductionsModel()` which returns the inferred triples,

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;
  - `getDeductionsModel()` which returns the inferred triples,
  - `getRawModel()` which returns the base triples,

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;
  - `getDeductionsModel()` which returns the inferred triples,
  - `getRawModel()` which returns the base triples,
  - `getReasoner()` which returns the RDFS reasoner,

Example I: Using a convenience method

A simple RDFS model

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```

method `createRDFSModel()` returns an `InfModel`

- An `InfModel` has a basic inference API, such as;
  - `getDeductionsModel()` which returns the inferred triples,
  - `getRawModel()` which returns the base triples,
  - `getReasoner()` which returns the RDFS reasoner,
  - `getDerivation(stmt)` which returns a trace of the derivation

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Example II: Using static methods in the registry

using `ModelFactory.createInfModel`

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

- we retain a reference to the reasoner,

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

- we retain a reference to the reasoner,
- that can be used to configure it

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

- we retain a reference to the reasoner,
- that can be used to configure it
    - e.g. to do backwards or forwards chaining

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

- we retain a reference to the reasoner,
- that can be used to configure it
  - e.g. to do backwards or forwards chaining
  - . . . mind you, not all reasoners can do both

Example II: Using static methods in the registry

using ModelFactory.createInfModel

```
Model sche = FileManager.get().loadModel(aURI);
Model dat = FileManager.get().loadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

Virtues of this approach:

- we retain a reference to the reasoner,
- that can be used to configure it
  - e.g. to do backwards or forwards chaining
  - . . . mind you, not all reasoners can do both
- similar for built-in and external reasoners alike

Conclusion

- We have seen that by modelling knowledge using the URIs in the RDF and RDFS vocabularies (e.g. `rdf:type`, `rdfs:subClassOf`, `rdfs:range`), the computer can derive *new* triples, that follows from our original triples.

Conclusion

- We have seen that by modelling knowledge using the URIs in the RDF and RDFS vocabularies (e.g. `rdf:type`, `rdfs:subClassOf`, `rdfs:range`), the computer can derive *new* triples, that follows from our original triples.
- The rules were very simple (e.g. if `x rdf:type A` and `A rdfs:subClassOf B` then `x rdf:type B`).

Conclusion

- We have seen that by modelling knowledge using the URIs in the RDF and RDFS vocabularies (e.g. `rdf:type`, `rdfs:subClassOf`, `rdfs:range`), the computer can derive *new* triples, that follows from our original triples.
- The rules were very simple (e.g. if `x rdf:type A` and `A rdfs:subClassOf B` then `x rdf:type B`).
- However, note that even the most complex mathematical proofs can be broken down into equally simple steps.

Conclusion

- We have seen that by modelling knowledge using the URIs in the RDF and RDFS vocabularies (e.g. `rdf:type`, `rdfs:subClassOf`, `rdfs:range`), the computer can derive *new* triples, that follows from our original triples.
- The rules were very simple (e.g. if `x rdf:type A` and `A rdfs:subClassOf B` then `x rdf:type B`).
- However, note that even the most complex mathematical proofs can be broken down into equally simple steps.
- It is when we have large knowledge bases and we can apply thousands or millions of derivations that the reasoning becomes really interesting.

Conclusion

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

Conclusion

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

That's it for today!

Remember the oblig!