

Unit 2 – RDF & SPARQL Semantics in detail

Axel Polleres

DERI, National University of Ireland, Galway

VU 184.268 Technologien für das Semantische Web

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
3. Semantics of SPARQL
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

RDF Graph – Formal Definitions

Let U be the set of URIs, B be the set of blank nodes (or “variables”), $L = L_t \cup L_p \cup L_{lang}$ be the set of literals (i.e., typed, plain, and plain lang-tagged)

An **RDF graph**, or simply a graph, is a set of RDF triples from $UB \times U \times UBL$.¹

A **vocabulary of a graph** V_G is the subset of UL mentioned in the graph.

A graph or triple without blank nodes is also called **ground**

¹We write short e.g. UBL for $U \cup B \cup L$.

RDF Graph – Example 1

Node: “edge labels” may appear as nodes and vice versa, e.g.

G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

G_2 :

```
ex:alice rdf:type foaf:Person.
```

G_3 :

```
_:alice foaf:knows ex:bob.  
_:alice foaf:name _:name.
```

G_4 :

```
_:alice foaf:knows ex:bob.  
_:alice foaf:name _:alice.
```

RDF Graph – Example 1

Node: “edge labels” may appear as nodes and vice versa, e.g.

G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

G_2 :

```
ex:alice rdf:type foaf:Person.
```

G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name .
```

G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice .
```

Again, we will occasionally write blank nodes as like this *Var*, to make clearer that actually they amount to existentially quantified variables.

RDF Graph – Example 2

That is also a valid RDF graph:

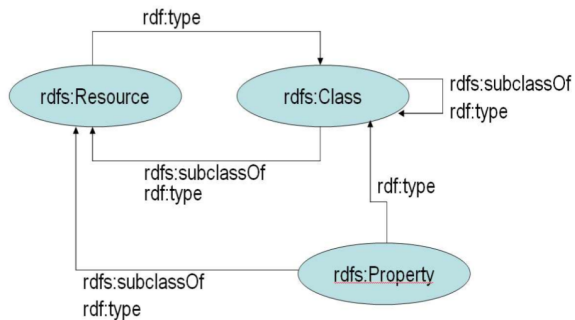
G_5 :

```
rdfs:Resource rdf:type rdfs:Class.  
rdfs:Property rdf:type rdfs:Resource.  
rdfs:Property rdf:subClassOf rdfs:Resource.  
rdfs:Property rdf:type rdfs:Class.  
rdfs:Class rdf:type rdfs:Resource.  
rdfs:Class rdf:type rdfs:Class.  
rdfs:Class rdf:subClassOf rdfs:Resource.  
rdfs:Class rdf:subClassOf rdfs:Class.
```

RDF Graph – Example 2

That is also a valid RDF graph:

G_5 :



RDF Graph – Example 3

Or that:

G_6 :

```
rdfs:subClassOf rdfs:subPropertyOf rdfs:Resource.  
rdfs:subClassOf rdfs:subPropertyOf rdfs:subPropertyOf.  
rdf:type rdfs:subPropertyOf rdfs:subClassOf.  
rdfs:subClassOf rdf:type owl:SymmetricProperty.
```


Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

A *proper instance* of a graph is an instance in which a blank node has been replaced by a name, or two blank nodes in the graph have been mapped into the same node in the instance.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

A *proper instance* of a graph is an instance in which a blank node has been replaced by a name, or two blank nodes in the graph have been mapped into the same node in the instance.

An RDF graph is *lean* if it has no instance which is a proper subgraph of the graph. Non-lean graphs have internal redundancy and express the same content as their lean subgraphs.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

A *proper instance* of a graph is an instance in which a blank node has been replaced by a name, or two blank nodes in the graph have been mapped into the same node in the instance.

An RDF graph is *lean* if it has no instance which is a proper subgraph of the graph. Non-lean graphs have internal redundancy and express the same content as their lean subgraphs.

Two graphs which differ only in the identity of their blank nodes, are considered to be *equivalent*.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

A *proper instance* of a graph is an instance in which a blank node has been replaced by a name, or two blank nodes in the graph have been mapped into the same node in the instance.

An RDF graph is *lean* if it has no instance which is a proper subgraph of the graph. Non-lean graphs have internal redundancy and express the same content as their lean subgraphs.

Two graphs which differ only in the identity of their blank nodes, are considered to be *equivalent*.

The *merge* of a set of graphs is obtained by renaming (“standardize apart”) blank nodes in each graph such that no blank nodes between any two graphs are in common and then taking the union of all triples, we write $G1 \uplus G2$ for the graph merge between two graphs $G1, G2$.

Definitions

Assume a blank node mapping $\mu : B \rightarrow UBL$.

By $\mu(G)$ we denote the graph obtained from G by replacing each blank node x with $\mu(x)$.

We call $\mu(G)$ an *instance* of G .

A *proper instance* of a graph is an instance in which a blank node has been replaced by a name, or two blank nodes in the graph have been mapped into the same node in the instance.

An RDF graph is *lean* if it has no instance which is a proper subgraph of the graph. Non-lean graphs have internal redundancy and express the same content as their lean subgraphs.

Two graphs which differ only in the identity of their blank nodes, are considered to be *equivalent*.

The *merge* of a set of graphs is obtained by renaming (“standardize apart”) blank nodes in each graph such that no blank nodes between any two graphs are in common and then taking the union of all triples, we write $G1 \uplus G2$ for the graph merge between two graphs $G1, G2$.

Lean and non-lean graphs: Examples

G_7 : non-lean

```
_:x foaf:knows ex:bob.
```

```
_:x foaf:knows _:y.
```

G_8 : lean

```
_:x foaf:knows ex:bob.
```

```
_:x foaf:knows _:x.
```

Why?

Lean and non-lean graphs: Examples

G_7 : non-lean

$\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \wedge \text{triple}(x, \text{knows}, y)$

G_8 : lean

$\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \wedge \text{triple}(x, \text{knows}, x)$

Becomes clear if we look at first-order “reading” of the RDF graph, where we treat blank nodes as existential variables and triples in a predicate *triple*. With this reading, one could say: $G'_7 = \{ _ : x \text{ foaf:knows } ex:\text{bob}. \} \models G_7$

Lean and non-lean graphs: Examples

 G_7 : non-lean $\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \models$ $\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \wedge \text{triple}(x, \text{knows}, y)$ G_8 : lean $\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \not\models$ $\exists x, y. \text{triple}(x, \text{knows}, \text{bob}) \wedge \text{triple}(x, \text{knows}, x)$

Becomes clear if we look at first-order “reading” of the RDF graph, where we treat blank nodes as existential variables and triples in a predicate *triple*. With this reading, one could say: $G'_7 = \{ _ : x \text{ foaf:knows } ex:\text{bob}. \} \models G_7$

We use first-order *entailment* here. Entailment is typically defined in terms of a model theory (interpretation, satisfaction, models)...

RDF has its own model theory!

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
3. Semantics of SPARQL
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

Model theoretic semantics – in general

A model theory is usually defined using the following “components”:

- Defining a notion of an **interpretation** I , consisting of separate interpretation functions
 - i.e., defining how are constants, variables and logical connectives, formulas being “interpreted” in a possible real world.
- A **satisfaction relation** between interpretations and theories (in our case graphs), written $I \models G$, which says:
 - I is an interpretation satisfying G , or a **model** of G
- An **entailment relation** between theories (in our case graphs), written $G \models G'$, which says
 - all models of G are also models of G'

Simple Interpretations1/4

*“**interpretation** I : ... i.e. how are constants, variables, predicates, formulas being “interpreted” in a possible real world.”*

What does that mean for RDF?

- RDF “constants” ... subjects, objects, i.e. UL
- RDF “variables” ... blank nodes, i.e. B
- RDF “predicates” ... predicates, i.e. U
- RDF “formulas” ... triples, graphs.

Simple Interpretations1/4

*“**interpretation** I : ... i.e. how are constants, variables, predicates, formulas being “interpreted” in a possible real world.”*

What does that mean for RDF?

- RDF “constants” ... subjects, objects, i.e. UL
- RDF “variables” ... blank nodes, i.e. B
- RDF “predicates” ... predicates, i.e. U
- RDF “formulas” ... triples, graphs.

Now **here** we have something unlike classical logic... URIs can actually need to be interpreted “as predicates” or “as constants” depending on where they appear in the graph.

To cater for that, RDF defines a very general notion of **interpretation**.

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
- 2 A set IP , called the set of properties of I ,
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
- 5 A mapping $IL : L_t \cap V$ into IR .
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - *called the domain or universe of I*
- 2 A set IP , called the set of properties of I ,
 - *not necessarily disjoint of IR !*
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - *this basically says, URIs can be both constants and predicates*
- 5 A mapping $IL : L_t \cap V$ into IR .
 - *typed literals are constants.*
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - *plain literals in RDF are special, they are always interpreted as themselves*

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - *called the domain or universe of I*
- 2 A set IP , called the set of properties of I ,
 - *not necessarily disjoint of IR !*
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - *this basically says, URIs can be both constants and predicates*
- 5 A mapping $IL : L_t \cap V$ into IR .
 - *typed literals are constants.*
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - *plain literals in RDF are special, they are always interpreted as themselves*

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - called the domain or universe of I
- 2 A set IP , called the set of properties of I ,
 - not necessarily disjoint of IR !
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - this basically says, URIs can be both constants and predicates
- 5 A mapping $IL : L_t \cap V$ into IR .
 - typed literals are constants.
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - plain literals in RDF are special, they are always interpreted as themselves

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - *called the domain or universe of I*
- 2 A set IP , called the set of properties of I ,
 - *not necessarily disjoint of IR !*
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - *this basically says, URIs can be both constants and predicates*
- 5 A mapping $IL : L_t \cap V$ into IR .
 - *typed literals are constants.*
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - *plain literals in RDF are special, they are always interpreted as themselves*

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - *called the domain or universe of I*
- 2 A set IP , called the set of properties of I ,
 - *not necessarily disjoint of IR !*
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - *this basically says, URIs can be both constants and predicates*
- 5 A mapping $IL : L_t \cap V$ into IR .
 - *typed literals are constants.*
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - *plain literals in RDF are special, they are always interpreted as themselves*

Simple Interpretations 2/4

A **simple interpretation** I over vocabulary V is a 6-tuple $I = \langle IR, IP, IEXT, IS, IL, LV \rangle$, s.t.

- 1 A non-empty set IR of resources.
 - *called the domain or universe of I*
- 2 A set IP , called the set of properties of I ,
 - *not necessarily disjoint of IR !*
- 3 A mapping $IEXT : IP \rightarrow 2^{(IR \times IR)}$, i.e. assigns a set of pairs $\langle x, y \rangle$ with $x, y \in IR$.
 - *intuitively, assigns a binary relation between subjects and objects to properties.*
- 4 A mapping $IS : U \cap V \rightarrow IR \cup IP$
 - *this basically says, URIs can be both constants and predicates*
- 5 A mapping $IL : L_t \cap V$ into IR .
 - *typed literals are constants.*
- 6 A distinguished subset $LV \subset IR$, called the set of **literal values**, which contains all the plain literals in V , i.e. $LV \subseteq L_p \cup L_{lang}$.
 - *plain literals in RDF are special, they are always interpreted as themselves*

Simple Interpretations 3/4

Interpreting **ground graphs** (i.e. without blank nodes):

■ Interpreting **constants**:

- if $e = \text{"aaa"} \in V \cap L_p$, then $I(e) = aaa \in LV$
- if $e = \text{"aaa"@ttt} \in V \cap L_{lang}$, then $I(e) = \langle aaa, ttt \rangle \in LV$
- if $e \in V \cap L_t$, then $I(e) = IL(e)$
- if $e \in V \cap U$, then $I(e) = IS(e)$

Simple Interpretations 3/4

Interpreting **ground graphs** (i.e. without blank nodes):

■ Interpreting **constants**:

- if $e = \text{"aaa"} \in V \cap L_p$, then $I(e) = aaa \in LV$
- if $e = \text{"aaa"@ttt} \in V \cap L_{lang}$, then $I(e) = \langle aaa, ttt \rangle \in LV$
- if $e \in V \cap L_t$, then $I(e) = IL(e)$
- if $e \in V \cap U$, then $I(e) = IS(e)$

■ Interpreting **ground triples**:

- if $t = s \text{ p } o$, is a ground triple, then
 - $I(t) = \text{true}$ if $s, p, o \in V \wedge I(p) \in IP \wedge \langle I(s), I(o) \rangle \in IEXT(I(p))$
 - $I(t) = \text{false}$, otherwise

Simple Interpretations 3/4

Interpreting **ground graphs** (i.e. without blank nodes):

■ Interpreting **constants**:

- if $e = \text{"aaa"} \in V \cap L_p$, then $I(e) = aaa \in LV$
- if $e = \text{"aaa"@ttt} \in V \cap L_{lang}$, then $I(e) = \langle aaa, ttt \rangle \in LV$
- if $e \in V \cap L_t$, then $I(e) = IL(e)$
- if $e \in V \cap U$, then $I(e) = IS(e)$

■ Interpreting **ground triples**:

- if $t = s \text{ p } o$, is a ground triple, then
 - $I(t) = \text{true}$ if $s, p, o \in V \wedge I(p) \in IP \wedge \langle I(s), I(o) \rangle \in IEXT(I(p))$
 - $I(t) = \text{false}$, otherwise

■ Interpreting **ground graphs**:

- if G is a ground RDF graph then $I(G) = \text{true}$ if and only if $I(t) = \text{true}$ for all triples $t \in G$, .

Simple Interpretations 3/4

Interpreting **ground graphs** (i.e. without blank nodes):

■ Interpreting **constants**:

- if $e = \text{"aaa"} \in V \cap L_p$, then $I(e) = aaa \in LV$
- if $e = \text{"aaa"@ttt} \in V \cap L_{lang}$, then $I(e) = \langle aaa, ttt \rangle \in LV$
- if $e \in V \cap L_t$, then $I(e) = IL(e)$
- if $e \in V \cap U$, then $I(e) = IS(e)$

■ Interpreting **ground triples**:

- if $t = s \text{ p } o$, is a ground triple, then
 - $I(t) = \text{true}$ if $s, p, o \in V \wedge I(p) \in IP \wedge \langle I(s), I(o) \rangle \in IEXT(I(p))$
 - $I(t) = \text{false}$, otherwise

■ Interpreting **ground graphs**:

- if G is a ground RDF graph then $I(G) = \text{true}$ if and only if $I(t) = \text{true}$ for all triples $t \in G$.

Satisfaction

If $I(G) = \text{true}$ we also say I **satisfies** G , written $I \models G$

Simple Interpretation – Example ground graphs

Take the following artificial vocabulary:

$$\{\text{ex : a}, \text{ex : b}, \text{ex : c}, \text{"whatever"}, \text{"whatever"} \wedge \text{ex : b}\}$$
$$IR = LV \cup \{1, 2\}$$
$$IP = \{1\}$$
$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$
$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$
$$IL(\text{"whatever"} \wedge \text{ex : b}) = 2$$

Simple Interpretation – Example ground graphs

Take the following artificial vocabulary:

$$\{\text{ex : a, ex : b, ex : c, "whatever", "whatever" \wedge \text{ex : b}}\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$

$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$

$$IL(\text{"whatever"} \wedge \text{ex : b}) = 2$$

G_9 :

ex:a ex:b ex:c .

ex:c ex:a ex:a .

ex:c ex:b ex:a .

ex:a ex:b "whatever" \wedge ex:b .

$I(G_9) = \text{true}$, i.e., $I \models G_9$:

Simple Interpretation – Example ground graphs

Take the following artificial vocabulary:

$$\{ex : a, ex : b, ex : c, \text{"whatever"}, \text{"whatever"}^{\wedge} ex : b\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

$$IEXT(1) = \{ \langle 1, 2 \rangle, \langle 2, 1 \rangle \}$$

$$IS(ex : a) = IS(ex : b) = 1, IS(ex : c) = 2$$

$$IL(\text{"whatever"}^{\wedge} ex : b) = 2$$

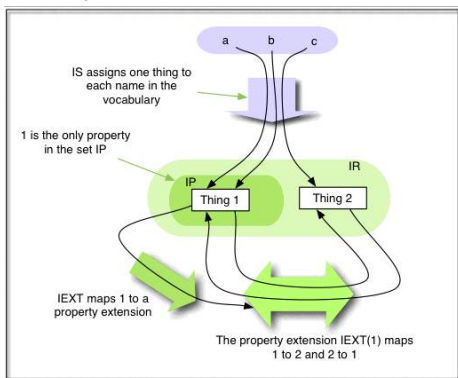
G_9 :

$ex:a \ ex:b \ ex:c \ .$

$ex:c \ ex:a \ ex:a \ .$

$ex:c \ ex:b \ ex:a \ .$

$ex:a \ ex:b \ \text{"whatever"}^{\wedge} ex:b \ .$



$I(G_9) = \text{true}$, i.e., $I \models G_9$:

Simple Interpretation – Example ground graphs

Take the following artificial vocabulary:

$$\{ex : a, ex : b, ex : c, \text{"whatever"}, \text{"whatever"} \wedge ex : b\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

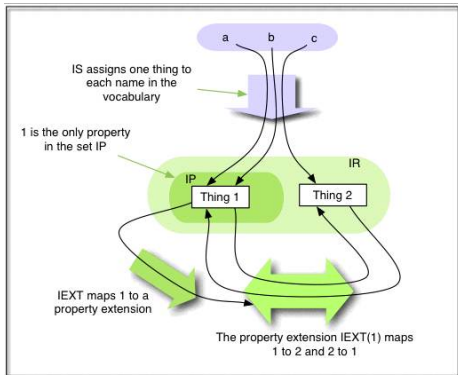
$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$

$$IS(ex : a) = IS(ex : b) = 1, IS(ex : c) = 2$$

$$IL(\text{"whatever"} \wedge ex : b) = 2$$

G'_9 :

$ex:a \quad ex:c \quad ex:b \quad .$
 $ex:a \quad ex:b \quad ex:b \quad .$
 $ex:c \quad ex:b \quad ex:c \quad .$
 $ex:a \quad ex:b \quad \text{"whatever"} \quad .$



$I(G'_9) = \text{false}$, i.e., I doesn't satisfy any triple in G'_9 :

Simple Interpretation – Example ground graphs

Take the following artificial vocabulary:

$$\{\text{ex : a}, \text{ex : b}, \text{ex : c}, \text{"whatever"}, \text{"whatever"} \wedge \text{ex : b}\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$

$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$

$$IL(\text{"whatever"} \wedge \text{ex : b}) = 2$$

G'_9 :

ex:a ex:c ex:b .	$IS(\text{ex : c}) = 2 \notin IP$
ex:a ex:b ex:b .	$\langle 1, 1 \rangle \notin IEXT(IS(\text{ex : b}))$
ex:c ex:b ex:c .	$\langle 2, 2 \rangle \notin IEXT(IS(\text{ex : b}))$
ex:a ex:b "whatever".	$\langle 1, \text{"whatever"} \rangle \notin IEXT(IS(\text{ex : b}))$

$I(G'_9) = \text{false}$, i.e., I doesn't satisfy any triple in G'_9 :

Simple Interpretations 4/4

Dealing with **blank nodes** is analogously to dealing with existential variables in first-order logic:

We call some function $A : B \rightarrow IR$ an **assignment**.

Given an interpretation I , and an assignment A , $[I + A]$ is defined just like I , except that it uses A to interpret blank nodes.

■ Interpreting non-ground graphs:

- if G is a non-ground RDF graph then $I(G) = \text{true}$ if and only if there exists an assignment A such that $[I + A](G) = \text{true}$.

Simple Interpretation – Example non-ground graphs

Same interpretation as before, artificial vocabulary:

$$\{\text{ex : a}, \text{ex : b}, \text{ex : c}, \text{"whatever"}, \text{"whatever"}^{\wedge} \text{ex : b}\}$$
$$IR = LV \cup \{1, 2\}$$
$$IP = \{1\}$$
$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$
$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$
$$IL(\text{"whatever"}^{\wedge} \text{ex : b}) = 2$$

Simple Interpretation – Example non-ground graphs

Same interpretation as before, artificial vocabulary:

$$\{\text{ex : a, ex : b, ex : c, "whatever", "whatever"}^{\wedge} \text{ex : b}\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$

$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$

$$IL(\text{"whatever"}^{\wedge} \text{ex : b}) = 2$$

G_{10} :

$$_ : x \langle \text{ex : a} \rangle \langle \text{ex : b} \rangle .$$

$$\langle \text{ex : c} \rangle \langle \text{ex : b} \rangle _ : y .$$

$I(G_{10}) = \text{true}$, i.e., $I \models G_{10}$:

E.g. take the assignment $A(x) = 2, A(y) = 1$

Simple Interpretation – Example non-ground graphs

Same interpretation as before, artificial vocabulary:

$$\{\text{ex : a, ex : b, ex : c, "whatever", "whatever"}^{\wedge} \text{ex : b}\}$$

$$IR = LV \cup \{1, 2\}$$

$$IP = \{1\}$$

$$IEXT(1) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$$

$$IS(\text{ex : a}) = IS(\text{ex : b}) = 1, IS(\text{ex : c}) = 2$$

$$IL(\text{"whatever"}^{\wedge} \text{ex : b}) = 2$$

G'_{10} :

$$_ : x \langle \text{ex : a} \rangle \langle \text{ex : b} \rangle .$$

$$\langle \text{ex : c} \rangle \langle \text{ex : b} \rangle _ : x .$$

$I(G'_{10}) = \text{false}$, i.e., $I \not\models G'_{10}$:

If A maps x to 1 then the first triple is false, and if it maps it to 2 then the second one.

Simple Entailment between RDF Graphs

The usual entailment relation as we know it from first-order theories:

Simple Entailment

An RDF graph G (simply) entails a graph E , written $G \models E$, if every interpretation which satisfies G also satisfies E

“Entailment is the key idea which connects model-theoretic semantics to real-world applications” [Hayes, 2004] . . . indeed, simple entailment is the key for SPARQL graph pattern matching.

Simple Entailment between RDF Graphs

The usual entailment relation as we know it from first-order theories:

Simple Entailment (for sets of graphs)

A set S of RDF graphs (simply) entails a graph E , written $S \models E$, if every interpretation which satisfies **every member of** S also satisfies E

“Entailment is the key idea which connects model-theoretic semantics to real-world applications” [Hayes, 2004] . . . indeed, simple entailment is the key for SPARQL graph pattern matching.

Simple Entailment - Properties

Merging lemma

The merge of a set S of RDF graphs is entailed by S , and entails every member of S , i.e.

$S \models \bigcup_{s \in S} s$ and $\bigcup_{s \in S} s \models s'$, where $s' \in S$.

Simple Entailment - Properties

Merging lemma

The merge of a set S of RDF graphs is entailed by S , and entails every member of S , i.e.

$S \models \biguplus_{s \in S} s$ and $\biguplus_{s \in S} s \models s'$, where $s' \in S$.

Recall the example from before:

G'_{10} :

$_ :x <ex:a> <ex:b> .$

$<ex:c> <ex:b> _ :x .$

This example shows the difference of union and merge:

The merge of each triple by itself taken as a singleton graph is **NOT** equivalent to G'_{10} !

(Recall the definition of merge: Obtained by “standardizing apart” blank nodes.)

Simple Entailment - Properties

Main result for simple RDF inference is:

Interpolation Lemma

S entails a graph E if and only if a subgraph of S is an instance of E .

Simple Entailment - Properties

Main result for simple RDF inference is:

Interpolation Lemma

S entails a graph E if and only if a subgraph of S is an instance of E .

What does this mean?

Recall: We call $\mu(G)$ an *instance* of G , where μ maps blank nodes to UBL .

So, you can test entailment $G \models G'$ by

- 1 guessing a mapping μ and
- 2 test whether $\mu(G') \subseteq G$

Simple Entailment - Properties

Main result for simple RDF inference is:

Interpolation Lemma

S entails a graph E if and only if a subgraph of S is an instance of E .

What does this mean?

Recall: We call $\mu(G)$ an *instance* of G , where μ maps blank nodes to UBL .

So, you can test entailment $G \models G'$ by

- 1 guessing a mapping μ and
- 2 test whether $\mu(G') \subseteq G$

Complexity

Simple entailment is NP-complete.

(proof in the end of the slides, time allowed)

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
_:alice foaf:knows ex:bob.  
_:alice foaf:name _:name.
```

 G_4 :

```
_:alice foaf:knows ex:bob.  
_:alice foaf:name _:alice.
```

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_1 \models G_3$:

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_1 \models G_3$:
$$\mu(\textit{Alice}) = \textit{ex : alice}, \mu(\textit{Name}) = \textit{" Alice"} \Rightarrow \mu(G_3) \subseteq G_1$$

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_1 \not\models G_4$:

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_1 \not\models G_4$:

no blank node mapping μ makes $\mu(G_4)$ a subset of G_1

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_3 \not\models G_4$:

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_3 \not\models G_4$:

no blank node mapping μ makes $\mu(G_4)$ a subset of G_3

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.  
Alice foaf:name Alice.
```

 $G_4 \models G_3$:

Simple Entailment - Examples 1/4

 G_1 :

```
ex:alice foaf:knows ex:bob.
ex:alice foaf:name "Alice".
foaf:knows rdfs:domain foaf:Person.
```

 G_3 :

```
Alice foaf:knows ex:bob.
Alice foaf:name Name :.
```

 G_4 :

```
Alice foaf:knows ex:bob.
Alice foaf:name Alice.
```

 $G_4 \models G_3$:
$$\mu(Alice) = Alice, \mu(Name) = Alice \Rightarrow \mu(G_3) \subseteq G_4$$

Simple Entailment - Examples 2/4²

G_7 : non-lean

X foaf:knows ex:bob.

X foaf:knows Y .

G_8 : lean

X foaf:knows ex:bob.

X foaf:knows X .

G'_7 : lean

X foaf:knows ex:bob.

G'_8 : lean

X foaf:knows X .

²draw on whiteboard

Simple Entailment - Examples 2/4²

G_7 : non-lean

X foaf:knows ex:bob.

X foaf:knows Y .

G_8 : lean

X foaf:knows ex:bob.

X foaf:knows X .

G'_7 : lean

X foaf:knows ex:bob.

G'_8 : lean

X foaf:knows X .

$G_7 \not\models G_8, G_7 \not\models G'_8$

²draw on whiteboard

Simple Entailment - Examples 2/4²

G_7 : non-lean

X foaf:knows ex:bob.

X foaf:knows Y .

G_8 : lean

X foaf:knows ex:bob.

X foaf:knows X .

G'_7 : lean

X foaf:knows ex:bob.

G'_8 : lean

X foaf:knows X .

$G_7 \not\models G_8, G_7 \not\models G'_8$

$G_8 \models G_7, G_7 \models G'_7$

²draw on whiteboard

Simple Entailment - Examples 2/4² G_7 : non-lean X foaf:knows ex:bob. X foaf:knows Y . G_8 : lean X foaf:knows ex:bob. X foaf:knows X . G'_7 : lean X foaf:knows ex:bob. G'_8 : lean X foaf:knows X . $G_7 \not\models G_8, G_7 \not\models G'_8$ $G_8 \models G_7, G_7 \models G'_7$ Finally: $G'_7 \models G_7$!!!! that confirms non-lean!

²draw on whiteboard

Simple Entailment - Examples 3/4

Now what about G_2 ?

G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

G_2 :

```
ex:alice rdf:type foaf:Person.
```

Obviously, no simple entailment: $G_1 \not\models G_2$!

Simple Entailment - Examples 3/4

Now what about G_2 ?

G_1 :

```
ex:alice foaf:knows ex:bob.  
ex:alice foaf:name "Alice".  
foaf:knows rdfs:domain foaf:Person.
```

G_2 :

```
ex:alice rdf:type foaf:Person.
```

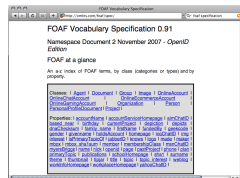
Obviously, no simple entailment: $G_1 \not\models G_2$!

Would need “special” interpretation of the `rdf:` and `rdfs:` vocabulary!

This is needed to interpret *ontologies*...

Recall from Unit1 – The FOAF ontology:

- **Properties:** name, knows, homepage, primaryTopic etc.
- **Classes:** Person, Agent, Document, Organisation, etc.
- **Relations:** e.g.
 - *Each Person is a Agent* (subclass)
 - *The img property is more specific than depiction* (subproperty)
 - *img is a relation between Persons and Images* (domain/range)
 - *knows is a relation between two Persons* (domain/range)

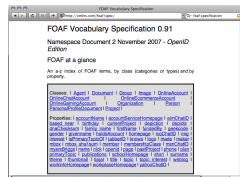


Recall from Unit1 – The FOAF ontology:

- **Properties:** name, knows, homepage, primaryTopic etc.
- **Classes:** Person, Agent, Document, Organisation, etc.
- **Relations:** e.g.

- *Each Person is a Agent* (subclass)
- *The img property is more specific than depiction* (subproperty)
- *img is a relation between Persons and Images* (domain/range)
- *knows is a relation between two Persons* (domain/range)
- *homepage denotes **unique** homepage of an Agent* (uniquely identifying property)

⋮



Simple Entailment - Examples 4/4

G'_1 :

`ex:alice foaf:knows ex:bob.`

`ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.`

G_{FOAF} : `<http://xmlns.com/foaf/0.1/>`

`foaf:knows rdfs:domain foaf:Person.`

`foaf:knows rdfs:range foaf:Person.`

`foaf:Person rdfs:subclassOf foaf:Agent.`

Simple Entailment - Examples 4/4

 G'_1 :`ex:alice foaf:knows ex:bob.``ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.` G_{FOAF} : `<http://xmlns.com/foaf/0.1/>``foaf:knows rdfs:domain foaf:Person.``foaf:knows rdfs:range foaf:Person.``foaf:Person rdfs:subclassOf foaf:Agent.`Intuitively, $G'_1 \uplus G_{FOAF}$ should entail: G'_2 :`ex:alice rdf:type foaf:Person.``ex:bob rdf:type foaf:Person.``ex:alice rdf:type foaf:Agent.``ex:bob rdf:type foaf:Agent.``ex:alice ex:age "30"^^xs:integer`

Simple Entailment - Examples 4/4

 G'_1 :`ex:alice foaf:knows ex:bob.``ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.` G_{FOAF} : `<http://xmlns.com/foaf/0.1/>``foaf:knows rdfs:domain foaf:Person.``foaf:knows rdfs:range foaf:Person.``foaf:Person rdfs:subclassOf foaf:Agent.`Intuitively, $G'_1 \uplus G_{FOAF}$ should entail: G'_2 :`ex:alice rdf:type foaf:Person. ... because the domain of knows is Person``ex:bob rdf:type foaf:Person.``ex:alice rdf:type foaf:Agent.``ex:bob rdf:type foaf:Agent.``ex:alice ex:age "30"^^xs:integer`

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments! [Hayes, 2004]. . .

Simple Entailment - Examples 4/4

 G'_1 :`ex:alice foaf:knows ex:bob.``ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.` G_{FOAF} : `<http://xmlns.com/foaf/0.1/>``foaf:knows rdfs:domain foaf:Person.``foaf:knows rdfs:range foaf:Person.``foaf:Person rdfs:subclassOf foaf:Agent.`Intuitively, $G'_1 \uplus G_{FOAF}$ should entail: G'_2 :`ex:alice rdf:type foaf:Person. ... because the domain of knows is Person``ex:bob rdf:type foaf:Person. ... because the range of knows is Person``ex:alice rdf:type foaf:Agent.``ex:bob rdf:type foaf:Agent.``ex:alice ex:age "30"^^xs:integer`

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments! [Hayes, 2004]...

Simple Entailment - Examples 4/4

 G'_1 :`ex:alice foaf:knows ex:bob.``ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.` G_{FOAF} : `<http://xmlns.com/foaf/0.1/>``foaf:knows rdfs:domain foaf:Person.``foaf:knows rdfs:range foaf:Person.``foaf:Person rdfs:subclassOf foaf:Agent.`Intuitively, $G'_1 \uplus G_{FOAF}$ should entail: G'_2 :`ex:alice rdf:type foaf:Person. ... because the domain of knows is Person``ex:bob rdf:type foaf:Person. ... because the range of knows is Person``ex:alice rdf:type foaf:Agent. ... because each Person is an Agent``ex:bob rdf:type foaf:Agent. ... because each Person is an Agent``ex:alice ex:age "30"^^xs:integer`

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments! [Hayes, 2004]. . .

Simple Entailment - Examples 4/4

 G'_1 :`ex:alice foaf:knows ex:bob.``ex:alice foaf:name "Alice". ex:alice ex:age "30.0"^^xs:decimal.` G_{FOAF} : `<http://xmlns.com/foaf/0.1/>``foaf:knows rdfs:domain foaf:Person.``foaf:knows rdfs:range foaf:Person.``foaf:Person rdfs:subclassOf foaf:Agent.`Intuitively, $G'_1 \uplus G_{FOAF}$ should entail: G'_2 :`ex:alice rdf:type foaf:Person. ... because the domain of knows is Person``ex:bob rdf:type foaf:Person. ... because the range of knows is Person``ex:alice rdf:type foaf:Agent. ... because each Person is an Agent``ex:bob rdf:type foaf:Agent. ... because each Person is an Agent``ex:alice ex:age "30"^^xs:integer ... simply because each 30.0 = 30`

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments! [Hayes, 2004]. . .

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary
 - e.g. imposes that $G'_1 \uplus G_{FOAF} \models \{ \text{ex:alice} \ \text{rdf:type} \ \text{foaf:Person} .\}$

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary
 - e.g. imposes that $G'_1 \uplus G_{FOAF} \models \{ \text{ex:alice} \ \text{rdf:type} \ \text{foaf:Person} .\}$
- D-entailment: Interpreting datatypes

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary
 - e.g. imposes that $G'_1 \uplus G_{FOAF} \models \{ \text{ex:alice} \ \text{rdf:type} \ \text{foaf:Person} .\}$
- D-entailment: Interpreting datatypes
 - e.g. imposing that in all interpretations that $"1" \wedge_{\text{xs:integer}}$ is interpreted the same as $"1.0" \wedge_{\text{xs:decimal}}$

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary
 - e.g. imposes that $G'_1 \uplus G_{FOAF} \models \{ \text{ex:alice} \ \text{rdf:type} \ \text{foaf:Person} .\}$
- D-entailment: Interpreting datatypes
 - e.g. imposing that in all interpretations that $"1" \wedge_{\text{xs:integer}}$ is interpreted the same as $"1.0" \wedge_{\text{xs:decimal}}$

RDF Entailment regimes beyond simple Entailment

The RDF semantics specification [Hayes, 2004] defines three refinements of simple interpretations and entailment relations which cover these entailments!:

- RDF-entailment: Interpreting the `rdf:` vocabulary
 - e.g. imposes that $\{s \ p \ o \ .\} \models p \ \text{rdf:type} \ \text{rdf:Property}$
- RDFS-entailment: Interpreting the `rdfs:` vocabulary
 - e.g. imposes that $G'_1 \uplus G_{FOAF} \models \{ \text{ex:alice} \ \text{rdf:type} \ \text{foaf:Person} .\}$
- D-entailment: Interpreting datatypes
 - e.g. imposing that in all interpretations that $"1" \wedge_{\text{xs:integer}}$ is interpreted the same as $"1.0" \wedge_{\text{xs:decimal}}$

More on that later! Now to the semantics of SPARQL...

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
- 3. Semantics of SPARQL**
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

Semantics of SPARQL

- The formal semantics of SPARQL [Prud'hommeaux and Seaborne, 2007] is based on simple Entailment
- i.e., no special interpretation of the RDFS vocabulary³
- semantics in the spec is defined in an operational way
- based on [Pérez *et al.*, 2006], who first defined a relational algebra for SPARQL
- some differences... similar issues as for SQL vs. relational algebra

For simplicity, we will focus on the formal semantics defined by [Pérez *et al.*, 2006] here, and only highlight the differences to the spec. semantics by examples.

³Not entirely true: the entailment regime is actually “parametric”, i.e. extensions allowed, more on that later.

SPARQL Semantics

Definition of a formal semantics of SPARQL:

http://www.polleres.net/sparqltutorial/ESWC2007_SPARQL_Tutorial_unit2a.pdf

Slides from *M. Arenas C. Gutierrez, J. Pérez*, ESWC 2007 Tutorial, Unit 2a.

The basic semantics is defined in slides **8–31**.

Notation used in SPARQL Semantics slides

Before we look into the semantics of Pérez et al. some notation:

They use an abbreviated version to write SPARQL query patterns:

$(((\{ t1 , t2 \} \text{ AND } t3) \text{ OPT } \{ t4 , t5 \}) \text{ AND } (t6 \text{ UNION } \{ t7 , t8 \}))$

stands for a pattern (or a sub-pattern) in the WHERE part of a query:

```
{  
  { { t1 . t2 } { t3 } OPTIONAL { t4 . t5 } }  
  { { { t6 } UNION { t7 . t8 } }  
}
```


Notation used in SPARQL Semantics slides

Definition Graph pattern:

- if t_1, t_2, \dots, t_n are triple patterns, then $\{t_1, t_2, \dots, t_n\}$ is a **basic graph pattern** (BGP)
- if P_1, P_2 are graph patterns, then
 - $(P_1 \text{ AND } P_1)$
 - $(P_1 \text{ UNION } P_1)$
 - $(P_1 \text{ OPT } P_1)$
 - $(P_1 \text{ FILTER } R)$, where R is a FILTER expression
 - $(G \text{ GRAPH } P_1)$, where G is a variable or in U

are graph patterns.

- Filter expressions: only `bound()`

SPARQL Semantics

Definition of a formal semantics of SPARQL:

http://www.polleres.net/sparqltutorial/ESWC2007_SPARQL_Tutorial_unit2a.pdf

Slides from *M. Arenas C. Gutierrez, J. Pérez*, ESWC 2007 Tutorial, Unit 2a.

The basic semantics is defined in slides 8–31.

Advice to the reader: SWITCH TO OTHER SLIDESET now ;-)

SPARQL Semantics

Definition of a formal semantics of SPARQL:

http://www.polleres.net/sparqltutorial/ESWC2007_SPARQL_Tutorial_unit2a.pdf

Slides from *M. Arenas C. Gutierrez, J. Pérez*, ESWC 2007 Tutorial, Unit 2a.

The basic semantics is defined in slides **8–31**.

Tricky parts:

- Blank nodes in Basic Graph patterns (treated slightly different in the spec)
- Blank nodes in CONSTRUCT queries
- Bag semantics, i.e. duplicates in solutions to SELECT queries
- Unsafe FILTERs (not treated in those slides in detail)
- FILTERs in OPTIONALs (not treated in those slides in detail)
- Higher Entailment regimes.

Blank nodes in Basic Graph patterns

The treatment in [**ESWC 2007 Tutorial, Unit 2a**, slide 32], gives a correct semantic specification for this corner case.

Blank nodes in Basic Graph patterns

The treatment in [ESWC 2007 Tutorial, Unit 2a, slide 32], gives a correct semantic specification for this corner case.

However, this is NOT according to the spec [Prud'hommeaux and Seaborne, 2007, Appendix A6]:

“The same blank node label may not be used in two separate basic graph patterns with a single query.”

This restriction allows us to treat all blank nodes just as variables, so no extra care for blank nodes is needed, normally.

Bottom line:

Preprocessing step 1:

Blank nodes in queries can be replaced equally using a unique, “fresh” variable for each blank node, the semantics of the query stays the same.

Blank nodes in Basic Graph patterns – Example

An example to illustrate this issue:

query9 : “*SELECT all persons known who have a homepage.*”

```
SELECT ?X
FROM <http://www.polleres.net/foaf.rdf>
WHERE { { _:b foaf:knows ?X } {?X foaf:homepage _:b } }
```

That one would not be compliant with the current spec!

Blank nodes in Basic Graph patterns – Example

An example to illustrate this issue:

query9b: “*SELECT all persons known who have a homepage.*”

```
SELECT ?X
FROM <http://www.polleres.net/foaf.rdf>
WHERE { _:b foaf:knows ?X . ?X foaf:homepage _:b }
```

Different meaning: *SELECT all persons known by their homepage.*

Blank nodes in Basic Graph patterns – Example

An example to illustrate this issue:

query9c: “*SELECT all persons known who have a homepage.*”

```
SELECT ?X
FROM <http://www.polleres.net/foaf.rdf>
WHERE { { _:b1 foaf:knows ?X } {?X foaf:homepage _:b2 } }
```

That one would work!

Blank nodes in Basic Graph patterns – Example

An example to illustrate this issue:

query9d: “*SELECT all persons known who have a homepage.*”

```
SELECT ?X
FROM <http://www.polleres.net/foaf.rdf>
WHERE { { ?B1 foaf:knows ?X } { ?X foaf:homepage ?B2 } }
```

That one is equivalent! Bnodes can be “treated” as variables

Blank nodes in CONSTRUCT queries

As shown in [ESWC 2007 Tutorial, Unit 2a, slide 31]:

- CONSTRUCT queries allow an arbitrary BGP which is used to construct a new graph as the RDF merge, of all solution mappings applied to the construct template.

This semantics ensures that

- blank nodes in CONSTRUCT pattern are treated correctly (fresh bnode for each solution)
- only valid RDF triples are constructed ($UB \times U \times UBL$)

Blank nodes in CONSTRUCT queries

As shown in [ESWC 2007 Tutorial, Unit 2a, slide 31]:

- CONSTRUCT queries allow an arbitrary BGP which is used to construct a new graph as the RDF merge, of all solution mappings applied to the construct template.

This semantics ensures that

- blank nodes in CONSTRUCT pattern are treated correctly (fresh bnode for each solution)
- only valid RDF triples are constructed ($UB \times U \times UBL$)

Some examples to understand this treatment. . .

Blank nodes in CONSTRUCT queries – Example 1

query10: “Anonymizing the people Alice knows”

G_{11} :

```
ex:alice foaf:knows ex:bob .
ex:alice foaf:knows ex:charles .
ex:alice foaf:name "Alice".
ex:alice foaf:knows _:d.
_:d foaf:name "Dorothy".
```

```
CONSTRUCT { ex:alice foaf:knows _:b }
FROM  $G_{11}$ 
WHERE { ex:alice foaf:knows ?X }
```

Result graph:

```
ex:alice foaf:knows _:genid1 .
ex:alice foaf:knows _:genid2 .
ex:alice foaf:knows _:genid3 .
```

The blank node labels, i.e., variable names, in the result graph can differ from implementation to implementation...

Blank nodes in CONSTRUCT queries – Example 1

query10: “Anonymizing the people Alice knows”

G_{11} :

```
ex:alice foaf:knows ex:bob .
ex:alice foaf:knows ex:charles .
ex:alice foaf:name "Alice".
ex:alice foaf:knows _:d.
_:d foaf:name "Dorothy".
```

```
CONSTRUCT { ex:alice foaf:knows _:b }
FROM  $G_{11}$ 
WHERE { ex:alice foaf:knows ?X }
```

Result graph:

```
ex:alice foaf:knows [] .
ex:alice foaf:knows [] .
ex:alice foaf:knows [] .
```

The blank node labels, i.e., variable names, in the result graph can differ from implementation to implementation...

... in Turtle syntax also possible here: **anonymous** blank nodes.

Blank nodes in CONSTRUCT queries – Example 2

query11 : “What is the node `ex:alice` connected to?”

G_{11} :

```
ex:alice foaf:knows ex:bob .
ex:alice foaf:knows ex:charles .
ex:alice foaf:name "Alice".
ex:alice foaf:knows _:d.
_:d foaf:name "Dorothy".
```

```
CONSTRUCT { ex:alice ex:connectsTo ?N }
FROM  $G_{11}$ 
WHERE { ex:alice ?P ?N }
```

Result graph:

```
ex:alice ex:connectsTo ex:bob .
ex:alice ex:connectsTo ex:charles .
ex:alice ex:connectsTo [] .
ex:alice ex:connectsTo "Alice".
```

Blank nodes in CONSTRUCT queries – Example 2

query11b: “What is the node `ex:alice` connected to?”

G_{11} :

```
ex:alice foaf:knows ex:bob .
ex:alice foaf:knows ex:charles .
ex:alice foaf:name "Alice".
ex:alice foaf:knows _:d.
_:d foaf:name "Dorothy".
```

```
CONSTRUCT { ?N ex:isConnectedTo ex:alice }
FROM  $G_{11}$ 
WHERE { ex:alice ?P ?N }
```

Result graph:

```
ex:bob ex:isConnectedTo ex:alice .
ex:charles ex:isConnectedTo ex:alice .
[] ex:isConnectedTo ex:alice .
```

In subject position, no literals allowed, the following “solution triple” is suppressed:

```
"Alice" ex:connectedTo ex:alice.
```

Note: the output graph can be non-lean!

Bag semantics

Shown in [ESWC 2007 Tutorial, Unit 2a, slide 33].

Essentially:

- SPARQL allows duplicate solutions, these may arise from
 - UNION patterns
 - Projections (i.e., variables projected away in the result form)

Some examples on that. . .

Bag semantics – Example 1: Duplicates from UNION

query12 : “Who knows bob OR Charles”

G'_{11} :

ex:alice foaf:knows ex:bob .

ex:alice foaf:knows ex:charles .

SELECT ?X

FROM G'_{11}

WHERE { { ?X foaf:knows ex:bob } }

UNION { ?X foaf:knows ex:charles} }

Result:

?X
ex:alice
ex:alice

Bag semantics – Example 1: Duplicates from UNION

query12b: “Who knows bob OR Charles”

G'_{11} :

ex:alice foaf:knows ex:bob .

ex:alice foaf:knows ex:charles .

SELECT DISTINCT ?X

FROM G'_{11}

WHERE { { ?X foaf:knows ex:bob } }

UNION { ?X foaf:knows ex:charles } }

Result:

?X
ex:alice

Bag semantics – Example 1: Duplicates from UNION

[query12c](#): “Who knows bob OR Charles”

G'_{11} :

```
ex:alice foaf:knows ex:bob .  
ex:alice foaf:knows ex:charles .
```

```
CONSTRUCT { ?X rdf:type ex:BobOrCharlesKnower }  
FROM  $G_{11}$   
WHERE { { ?X foaf:knows ex:bob }  
        UNION { ?X foaf:knows ex:charles} }
```

Result graph:

```
ex:alice rdf:type ex:BobOrCharlesKnower .
```

Bag semantics – Example 1: Duplicates from UNION

query12d: “Who knows bob OR Charles”

G'_{11} :

```
ex:alice foaf:knows ex:bob .  
ex:alice foaf:knows ex:charles .
```

```
CONSTRUCT { _:X rdf:type ex:BobOrCharlesKnower }  
FROM  $G_{11}$   
WHERE { { ?X foaf:knows ex:bob }  
        UNION { ?X foaf:knows ex:charles} }
```

Result graph:

```
_:genid1 rdf:type ex:BobOrCharlesKnower .  
_:genid2 rdf:type ex:BobOrCharlesKnower .
```

Note here: Blank nodes in CONSTRUCT also are affected by duplicate solutions!

Bag semantics – Example 1: Duplicates from UNION

query12d: “Who knows bob OR Charles”

G'_{11} :

```
ex:alice foaf:knows ex:bob .  
ex:alice foaf:knows ex:charles .
```

```
CONSTRUCT { _:Y rdf:type ex:BobOrCharlesKawner }  
FROM  $G'_{11}$   
WHERE { { ?X foaf:knows ex:bob }  
        UNION { ?X foaf:knows ex:charles} }
```

Result graph:

```
_:genid1 rdf:type ex:BobOrCharlesKawner .  
_:genid2 rdf:type ex:BobOrCharlesKawner .
```

Note here: Blank nodes in CONSTRUCT also are affected by duplicate solutions!

The blank node id in a construct template is completely irrelevant

Bag semantics – Example 1: Duplicates from UNION

query12d: “Who knows bob OR Charles”

G'_{11} :

```
ex:alice foaf:knows ex:bob .  
ex:alice foaf:knows ex:charles .
```

```
CONSTRUCT { [] rdf:type ex:BobOrCharlesKnower }  
FROM  $G'_{11}$   
WHERE { { ?X foaf:knows ex:bob }  
        UNION { ?X foaf:knows ex:charles} }
```

Result graph:

```
_:genid1 rdf:type ex:BobOrCharlesKnower .  
_:genid2 rdf:type ex:BobOrCharlesKnower .
```

Note here: Blank nodes in CONSTRUCT also are affected by duplicate solutions!

The blank node id in a construct template is completely irrelevant

Bag semantics – Example 2: Duplicates from projection

query12e: “Who knows whom?”

G'_{11} :

ex:alice foaf:knows ex:bob .

ex:alice foaf:knows ex:charles .

SELECT ?X ?Y

FROM G'_{11}

WHERE { ?X foaf:knows ?Y }

Result:

?X	?Y
ex:alice	ex:bob
ex:alice	ex:charles

Bag semantics – Example 2: Duplicates from projection

query12f: “Who knows somebody?”

G'_{11} :

ex:alice foaf:knows ex:bob .

ex:alice foaf:knows ex:charles .

SELECT ?X

FROM G_{11}

WHERE { ?X foaf:knows ?Y }

Result:

?X
ex:alice
ex:alice

Bag semantics – Example 2: Duplicates from projection

query12g: “Who knows somebody?”

G'_{11} :

ex:alice foaf:knows ex:bob .

ex:alice foaf:knows ex:charles .

SELECT ?X

FROM G'_{11}

WHERE { ?X foaf:knows [] }

Result:

?X
ex:alice
ex:alice

Unsafe FILTERs and Errors in FILTERs

For patterns of the form (P FILTER R)

- variables, appearing in R but not in P are problematic.
- complex filter expression, i.e. if R uses \neg , \wedge , \vee follow a 3-valued logic (\top , \perp , err)

Unsafe FILTER expression

Given a pattern (P FILTER R) we call R **unsafe** if it contains a variable not occurring in P .

Unsafe FILTERs – Examples

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .  
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40 .
```

query13:

```
SELECT ?X ?H  
WHERE { ?X rdf:type foaf:Person. ?X foaf:homepage ?H .  
        ?X ex:age ?A FILTER( ?A > 30 ) }
```

Result:

?X	?H
ex:charles	ex:hp2

Unsafe FILTERs – Examples

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .  
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40 .
```

query13b:

```
SELECT ?X ?H  
WHERE { ?X rdf:type foaf:Person. ?X foaf:homepage ?H .  
        FILTER( ?A > 30 ) }
```

Result:

?X	?H
----	----

“Unsafe” variables in FILTERs just have to be treated as **unbound**, so the FILTER evaluates to “*unbound* > 30” which is an error, thus the FILTER expression always fails, independent of the input graph.

Unsafe FILTERs – Examples

Note: unbound variables do not always yield the overall FILTER expression to fail!

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40 .
```

query13c:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person. ?X foaf:homepage ?H
       FILTER( ! bound(?A) ) }
```

Result:

?X	?H
ex:bob	ex:hp1
ex:charles	ex:hp2

That one is no problem!

Unsafe FILTERs – Examples

Note: unbound variables do not always yield the overall FILTER expression to fail!

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40 .
```

query13c:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person. ?X foaf:homepage ?H
        FILTER( ! bound(?A) ) }
```

Result:

?X	?H
ex:bob	ex:hp1
ex:charles	ex:hp2

That one is no problem!

However, there are **exceptions** concerning unsafe FILTERs...

“Unsafe” FILTERs – Exception 1: Filters within a group

[Prud'hommeaux and Seaborne, 2007, Section 5.2.2] “A *constraint*, expressed by the keyword *FILTER*, is a restriction on solutions over the whole group in which the filter appears.”

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40.
```

query14:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person.   FILTER( isIRI(?H) ) ?X foaf:homepage ?H }
```

Result:

?X	?H
ex:bob	ex:hp1
ex:charles	ex:hp2

“Unsafe” FILTERs – Exception 1: Filters within a group

[Prud'hommeaux and Seaborne, 2007, Section 5.2.2] “A *constraint*, expressed by the keyword *FILTER*, is a restriction on solutions over the whole group in which the filter appears.”

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40.
```

query14b:

```
SELECT ?X ?H
WHERE { { ?X rdf:type foaf:Person. } FILTER( isIRI(?H) ) { ?X foaf:homepage ?H } }
```

Result:

?X	?H
ex:bob	ex:hp1
ex:charles	ex:hp2

“Unsafe” FILTERs – Exception 1: Filters within a group

[Prud'hommeaux and Seaborne, 2007, Section 5.2.2] “A *constraint*, expressed by the keyword *FILTER*, is a restriction on solutions over the whole group in which the filter appears.”

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40.
```

query14c:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person.           ?X foaf:homepage ?H FILTER( isIRI(?H) ) }
```

Result:

?X	?H
ex:bob	ex:hp1
ex:charles	ex:hp2

“Unsafe” FILTERs – Exception 1: Filters within a group

[Prud'hommeaux and Seaborne, 2007, Section 5.2.2] “A *constraint*, expressed by the keyword *FILTER*, is a restriction on solutions over the whole group in which the filter appears.”

G_{12} :

```
ex:bob a foaf:Person; foaf:homepage ex:hp1; ex:age 20 .
```

```
ex:charles a foaf:Person; foaf:homepage ex:hp2; ex:age 40.
```

query14d: **BUT:**

```
SELECT ?X ?H
WHERE { { ?X rdf:type foaf:Person.      FILTER( isIRI(?H) ) } ?X foaf:homepage ?H }
```

Result:

?X	?H
----	----

“Unsafe” FILTERs – Exception 1: Filters within a group

Actually, this is not really an “exception”, but just a matter of translation to the relational syntax, where FILTERs are always moved last and are concatenated.

“Unsafe” FILTERs – Exception 1: Filters within a group

Actually, this is not really an “exception”, but just a matter of translation to the relational syntax, where FILTERs are always moved last and are concatenated.

Normalization, i.e. exhaustive application of the following rules:

- $P1 \text{ FILTER } R \text{ } P2 \Rightarrow ((P1 \text{ AND } P2) \text{ FILTER } R)$
- $(P \text{ FILTER } R1) \text{ FILTER } R2 \Rightarrow (P \text{ FILTER } (R1 \wedge R2))$

“Unsafe” FILTERs – Exception 1: Filters within a group

Actually, this is not really an “exception”, but just a matter of translation to the relational syntax, where FILTERs are always moved last and are concatenated.

Normalization, i.e. exhaustive application of the following rules:

- $P1 \text{ FILTER } R \text{ } P2 \Rightarrow ((P1 \text{ AND } P2) \text{ FILTER } R)$
- $(P \text{ FILTER } R1) \text{ FILTER } R2 \Rightarrow (P \text{ FILTER } (R1 \wedge R2))$

Intuitively: move FILTERs always to the end within a group, before evaluating the semantics.

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

“select Persons, and homepages only of those older than 30”

G_{12} as before.

query15:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person.  ?X ex:age ?A
       OPTIONAL { ?X foaf:homepage ?H FILTER( ?A > 30 ) } }
```

Result:

?X	?H
ex:bob	
ex:charles	ex:hp2

$\{\mu_1 = \{X \rightarrow \text{bob}\}, \mu_2 = \{X \rightarrow \text{charles}, H \rightarrow \text{hp2}\}$

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

“select Persons, and homepages only of those older than 30”

G_{12} as before.

query15:

```
SELECT ?X ?H
WHERE { ?X rdf:type foaf:Person.  ?X ex:age ?A
       OPTIONAL { ?X foaf:homepage ?H FILTER( ?A > 30 ) } }
```

Result:

?X	?H
ex:bob	
ex:charles	ex:hp2

$\{\mu_1 = \{X \rightarrow \text{bob}\}, \mu_2 = \{X \rightarrow \text{charles}, H \rightarrow \text{hp2}\}$

In the original semantics of [Pérez *et al.*, 2006] this would never return any homepage, since the FILTER is considered unsafe, i.e. would fail, and the OPTIONAL pattern would never return any solutions.

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this?

Semantics in [ESWC 2007 Tutorial, Unit 2a, slide 3] for OPTIONALs:

Semantics of OPT

$$[[P_1 \text{ OPT } P_2]]_G = [[P_1]]_G \supset \bowtie [[P_2]]_G = ([[P_1]]_G \bowtie [[P_2]]_G) \cup ([[P_1]]_G \setminus [[P_2]]_G)$$

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this?

Semantics in [ESWC 2007 Tutorial, Unit 2a, slide 3] for OPTIONALs:

Semantics of OPT

$$[[(P_1 \text{ OPT } P_2)]]_G = [[P_1]]_G \sqsupset \bowtie [[P_2]]_G = ([[P_1]]_G \bowtie [[P_2]]_G) \cup ([[P_1]]_G \setminus [[P_2]]_G)$$

Problem: This def. assumes a **compositional semantics**, i.e., that $[[(P_1 \circ P_2)]]$ can always be defined **modularly** with respect to $[[P_1]]$ and $[[P_2]] \dots$

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this?

Semantics in [ESWC 2007 Tutorial, Unit 2a, slide 3] for OPTIONALs:

Semantics of OPT

$$[[(P_1 \text{ OPT } P_2)]]_G = [[P_1]]_G \sqsupset \bowtie [[P_2]]_G = ([[P_1]]_G \bowtie [[P_2]]_G) \cup ([[P_1]]_G \setminus [[P_2]]_G)$$

Problem: This def. assumes a **compositional semantics**, i.e., that $[[(P_1 \circ P_2)]]$ can always be defined **modularly** with respect to $[[P_1]]$ and $[[P_2]]$...

... Unfortunately, for $[[(P_1 \text{ OPT } P_2)]]$, if there is an unsafe FILTER in P_2 , the official SPARQL semantics is **not** compositional!

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this? Solution:

[Prud'hommeaux and Seaborne, 2007, Section 12.5] rather says:

Semantics of OPT

A mapping μ is in $[[(P_1 \text{ OPT } (P_2 \text{ FILTER } R))]]_G$ if and only if:

- $\mu = \mu_1 \cup \mu_2$, s.t. $\mu_1 \in ([[P_1]]_G)$ and $\mu_2 \in ([[P_2]]_G)$ are compatible, and μ satisfies R , or
- $\mu \in ([[P_1]]_G)$ and there is no compatible $\mu_2 \in ([[P_2]]_G)$ for μ , or
- $\mu \in ([[P_1]]_G)$ and for any compatible $\mu_2 \in ([[P_2]]_G)$, $\mu \cup \mu_2$ does not satisfy R .

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this? Solution:

[Prud'hommeaux and Seaborne, 2007, Section 12.5] rather says:

Semantics of OPT

A mapping μ is in $[[(P_1 \text{ OPT } (P_2 \text{ FILTER } R))]]_G$ if and only if:

- $\mu = \mu_1 \cup \mu_2$, s.t. $\mu_1 \in ([[P_1]]_G)$ and $\mu_2 \in ([[P_2]]_G)$ are compatible, and μ satisfies R , or
- $\mu \in ([[P_1]]_G)$ and there is no compatible $\mu_2 \in ([[P_2]]_G)$ for μ , or
- $\mu \in ([[P_1]]_G)$ and for any compatible $\mu_2 \in ([[P_2]]_G)$, $\mu \cup \mu_2$ does not satisfy R .

Important: As opposed to the compositional definition, now the definition of OPT has 3 components!

Unsafe FILTERs – Exception 2: FILTERs in OPTIONALs

How to fix this? Solution:

[Prud'hommeaux and Seaborne, 2007, Section 12.5] rather says:

Semantics of OPT

A mapping μ is in $[[(P_1 \text{ OPT } (P_2 \text{ FILTER } R))]_G$ if and only if:

- $\mu = \mu_1 \cup \mu_2$, s.t. $\mu_1 \in ([[P_1]]_G$ and $\mu_2 \in [[P_2]]_G$ are compatible, and μ satisfies R , or
- $\mu \in ([[P_1]]_G$ and there is no compatible $\mu_2 \in [[P_2]]_G$ for μ , or
- $\mu \in ([[P_1]]_G$ and for any compatible $\mu_2 \in [[P_2]]_G$, $\mu \cup \mu_2$ does not satisfy R .

Important: As opposed to the compositional definition, now the definition of OPT has 3 components!

Positive message: Can be emulated with SPARQL with safe FILTERs by a rewriting! [Angles and Gutierrez, 2008, Theorem 1].

Complex FILTERs

Attention! \wedge (&&), \vee (||), \neg (!), in SPARQL FILTERs are not evaluated with respect to the usual boolean algebra, but (similar to SQL) in a 3-valued logic.

e.g. $eval("40" \wedge xs:integer > "30" \wedge xs:integer) = true$,

$eval("20" \wedge xs:integer > "20" \wedge xs:integer) = false$,

$eval("old" > "30" \wedge xs:integer) = err$

(cf. [query16](#)), similar for [query13b](#) before

Complex FILTERs

Attention! \wedge (&&), \vee (||), \neg (!), in SPARQL FILTERs are not evaluated with respect to the usual boolean algebra, but (similar to SQL) in a 3-valued logic.

e.g. $eval("40" \wedge xs:integer > "30" \wedge xs:integer) = true$,

$eval("20" \wedge xs:integer > "20" \wedge xs:integer) = false$,

$eval("old" > "30" \wedge xs:integer) = err$

(cf. [query16](#)), similar for [query13b](#) before

Since a FILTER constraint R can result not only in *true* and *false*, but also in *err*, the semantics of FILTERs has to reflect that:

$eval(R)$:

R	$\neg R$
<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>
<i>err</i>	<i>err</i>

R_1	R_2	$R_1 \wedge R_2$
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>err</i>	<i>err</i>
<i>err</i>	<i>true</i>	<i>err</i>
<i>false</i>	<i>err</i>	<i>false</i>
<i>err</i>	<i>false</i>	<i>false</i>
<i>err</i>	<i>err</i>	<i>err</i>

R_1	R_2	$R_1 \vee R_2$
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>err</i>	<i>true</i>
<i>err</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>err</i>	<i>err</i>
<i>err</i>	<i>false</i>	<i>err</i>
<i>err</i>	<i>err</i>	<i>err</i>

Complex FILTERs – Example

query17:

```
SELECT ?X ?A
WHERE { ?X rdf:type foaf:Person.  ?X ex:age ?A
        FILTER ( !(?A > ?X ) && (?A > 20) ) }
```


Complex FILTERs – Example

query17:

```
SELECT ?X ?A
WHERE { ?X rdf:type foaf:Person.  ?X ex:age ?A
        FILTER ( !(?A > ?X ) && (?A > 20) ) }
```

This will not return a result, because comparison of a literal and a resource yields *err*:

$$\text{eval}(\neg \text{err} \wedge \text{true}) = \text{err}$$

$$\text{eval}(\neg \text{err} \wedge \text{false}) = \text{false}$$

$$\text{eval}(\neg \text{err} \wedge \text{err}) = \text{err}$$

Complex FILTERs – Example

query17b:

```
SELECT ?X ?A
WHERE { ?X rdf:type foaf:Person.  ?X ex:age ?A
       FILTER ( !( (?A > ?X ) && (?A > 20) ) ) }
```

This one works for $\mu = \{?X \rightarrow \text{ex:bob}, ?A \rightarrow 20\}$:

?X	?A
ex:bob	20

$$\text{eval}(\neg(\text{err} \wedge \text{false}) = \text{true}$$

Higher Entailment regimes

All the semantics of SPARQL relies on the definition of BGP matching being built “on-top” of RDF simple entailment.

What if we want to consider “higher” entailments (RDF-, RDFS-, D-, OWL-entailment)?

- As we will see, some problems with that. . .
- We will talk about this, after we have talked about RDF- RDFS- and OWL-Entailment.

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
3. Semantics of SPARQL
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

Simple RDF Entailment is NP-complete: Membership

Recall, we had that before already: We can test entailment $G \models G'$ by

- 1 guessing a mapping μ and
- 2 test whether $\mu(G') \subseteq G$ (this is obviously polynomial)

Membership in NP - **done**

Simple RDF Entailment is NP-complete: Hardness

To proof hardness we have to reduce another NP-hard problem to RDF entailment (in polynomial time). Let's “adapt” the proof from [Chandra and Merlin, 1977].

Simple RDF Entailment is NP-complete: Hardness

To proof hardness we have to reduce another NP-hard problem to RDF entailment (in polynomial time). Let's “adapt” the proof from [Chandra and Merlin, 1977].

3-colorability: Given an undirected Graph G_r , can all nodes be colored with 3 colors **red**, **green**, **blue** without two adjacent nodes having the same color?

Simple RDF Entailment is NP-complete: Hardness

To proof hardness we have to reduce another NP-hard problem to RDF entailment (in polynomial time). Let's "adapt" the proof from [Chandra and Merlin, 1977].

3-colorability: Given an undirected Graph G_r , can all nodes be colored with 3 colors **red**, **green**, **blue** without two adjacent nodes having the same color?

Reduction (the "trick" is we have to convert an undirected to a directed RDF graph):

- Graph G_1 : simply encodes all "allowed" edges:
:red :edge :green. :green :edge :red.
:green :edge :blue. :blue :edge :green.
:blue :edge :red. :red :edge :blue.
- Graph G_2 : for each $(node_1, node_2) \in G_r$ we add two triples:
_:n1 :edge _:n2. _:n2 :edge _:n1.
to the graph G_2 , i.e, we model the nodes as blank nodes.

Simple RDF Entailment is NP-complete: Hardness

To proof hardness we have to reduce another NP-hard problem to RDF entailment (in polynomial time). Let's "adapt" the proof from [Chandra and Merlin, 1977].

3-colorability: Given an undirected Graph Gr , can all nodes be colored with 3 colors **red**, **green**, **blue** without two adjacent nodes having the same color?

Reduction (the "trick" is we have to convert an undirected to a directed RDF graph):

- Graph G_1 : simply encodes all "allowed" edges:
 - red** :edge **green**. **green** :edge **red**.
 - green** :edge **blue**. **blue** :edge **green**.
 - blue** :edge **red**. **red** :edge **blue**.
- Graph G_2 : for each $(node_1, node_2) \in Gr$ we add two triples:
 - `_:n1 :edge _:n2.` `_:n2 :edge _:n1.`
 to the graph G_2 , i.e, we model the nodes as blank nodes.

Now, it is easy to see that:

Proposition

Gr is 3-colorably if and only if $G_1 \models G_2$

Complexity of SPARQL evaluation

Time allowed (this is NOT subject of the exam!)

http://www.polleres.net/sparqltutorial/ESWC2007_SPARQL_Tutorial_unit4.pdf

Slides from M. Arenas C. Gutierrez, J. Pérez, ESWC 2007 Tutorial
Complexity of SPARQL evaluation, slides: 40–46.

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
3. Semantics of SPARQL
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

From SPARQL to Rules

Time allowed (this is NOT subject of the exam!)

<http://www.polleres.net/presentations/20080109talk-cosenza.pdf>

Unit Outline

1. RDF Graph – Formal Definitions
2. RDF Interpretations and Simple Entailment
3. Semantics of SPARQL
4. Complexity of simple RDF entailment and SPARQL
5. From SPARQL to Rules
6. Simple RDF Entailment acyclic graphs

Simple Entailment is polynomial for ground and acyclic graphs

Time allowed (this is NOT subject of the exam!)


http://www.polleres.net/presentations/20080605dRDF_ESWC2008.pdf

Recommended Reading

- [Gutiérrez *et al.*, 2004], excellent article on the logical foundations of RDF
- [de Bruijn *et al.*, 2005], relating RDF entailment to normal first-order logic.
- [Pérez *et al.*, 2006], SPARQL Semantics.

A bit more tough reading (specs), but also recommended:

- [Hayes, 2004, Sections 1–2], official RDF semantics specification.
- [Prud'hommeaux and Seaborne, 2007, Section 12], official SPARQL semantics specification.

-  Renzo Angles and Claudio Gutierrez.
The expressive power of sparql.
In *International Semantic Web Conference (ISWC 2008)*, pages 114–129, 2008.
-  A. K. Chandra and P. M. Merlin.
Optimal Implementation of Conjunctive Queries in Relational Data Bases.
In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.
-  Jos de Bruijn, Enrico Franconi, and Sergio Tessaris.
Logical reconstruction of normative RDF.
In *OWL: Experiences and Directions Workshop (OWLED-2005)*, Galway, Ireland, November 2005.
-  Claudio Gutiérrez, Carlos A. Hurtado, and Alberto O. Mendelzon.
Foundations of semantic web databases.
In *PODS*, pages 95–106, 2004.
-  P. Hayes.
RDF semantics, 2004.
<http://www.w3.org/TR/rdf-mt/>.
-  Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez.
Semantics and complexity of sparql.
In *International Semantic Web Conference (ISWC 2006)*, pages 30–43, 2006.
-  SPARQL Query Language for RDF, January 2007.

W3C Recommendation, available at

<http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.