RDF

Semantic Web

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

[Tim Berners-Lee et al. 2001.]

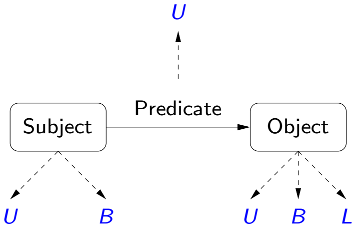
Specific Goals:

- Build a description language with standard semantics
- Make semantics machine-processable and understandable
- Incorporate logical infrastructure to reason about resources
- W3C Proposal: Resource Description Framework (RDF)

RDF in a nutshell

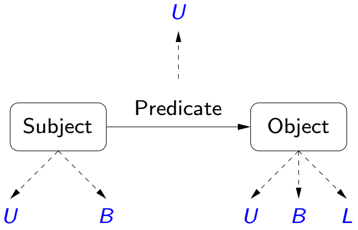
- RDF is the W3C proposal framework for representing information in the Web
- Abstract syntax based on directed labeled graph
- Schema definition language (RDFS): Define new vocabulary (typing, inheritance of classes and properties)
- Extensible URI-based vocabulary
- Formal semantics

RDF formal model



$$
\begin{aligned}
U &= \text{set of Uris} \\
B &= \text{set of Blank nodes} \\
L &= \text{set of Literals}
\end{aligned}
$$

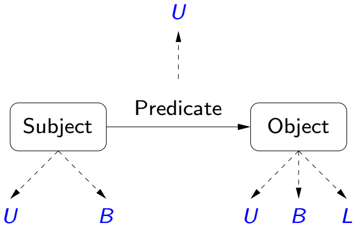
RDF formal model



$$U = \text{set of Uris}$$
$$B = \text{set of Blank nodes}$$
$$L = \text{set of Literals}$$

$(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple

RDF formal model



U = set of Uris

B = set of Blank nodes

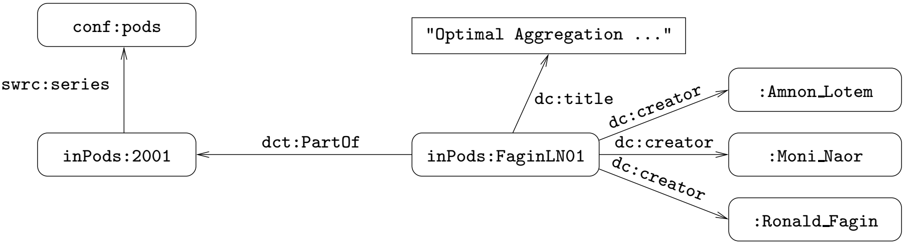
L = set of Literals

$(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple

A set of RDF triples is called an RDF graph

An example of an RDF graph: DBLP

```
      : <http://dblp.l3s.de/d2r/resource/authors/>
   conf: <http://dblp.l3s.de/d2r/resource/conferences/>
 inPods: <http://dblp.l3s.de/d2r/resource/publications/conf/pods/>
   swrc: <http://swrc.ontoware.org/ontology#>
     dc: <http://purl.org/dc/elements/1.1/>
    dct: <http://purl.org/dc/terms/>
```



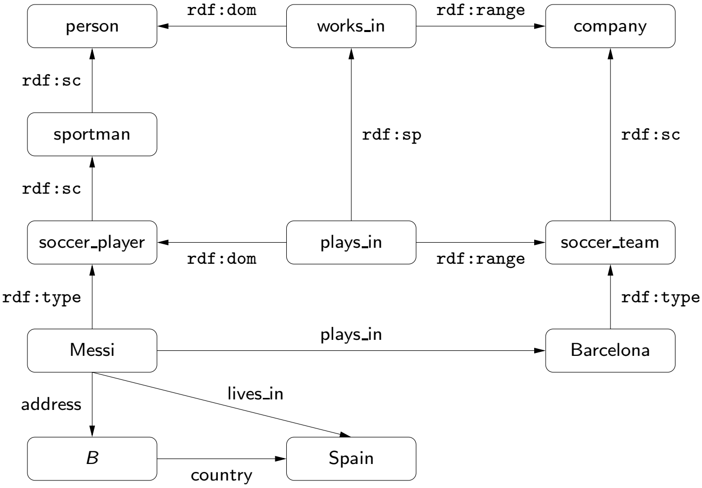
An example of a URI

http://dblp.l3s.de/d2r/resource/conferences/pods

URI can be used for any abstract resource

`http://dblp.l3s.de/d2r/page/authors/Ronald_Fagin`

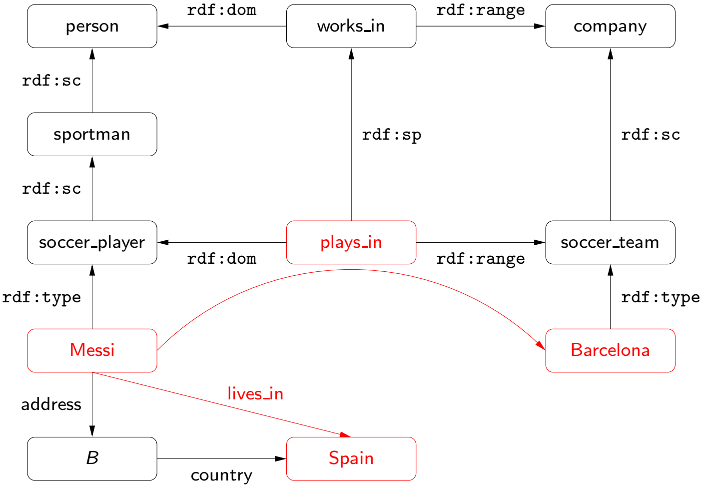
RDF: Another example



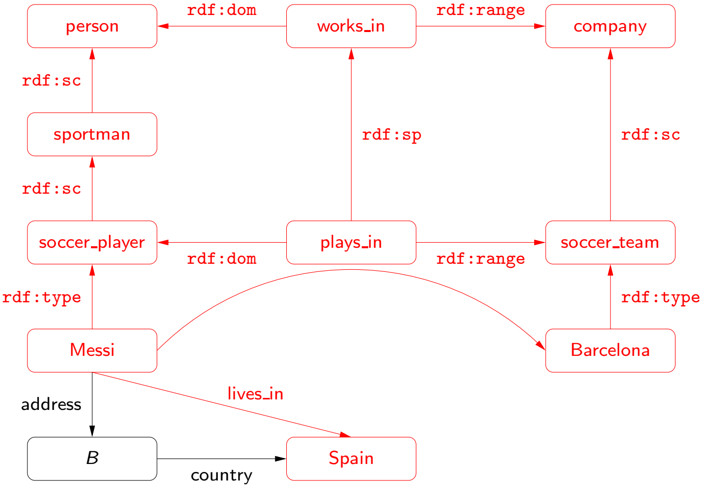
Some peculiarities of the RDF data model

- *Existential variables* as datavalues (null values)
- Built-in vocabulary with fixed semantics (RDFS)
- Graph model where nodes may also be edge labels

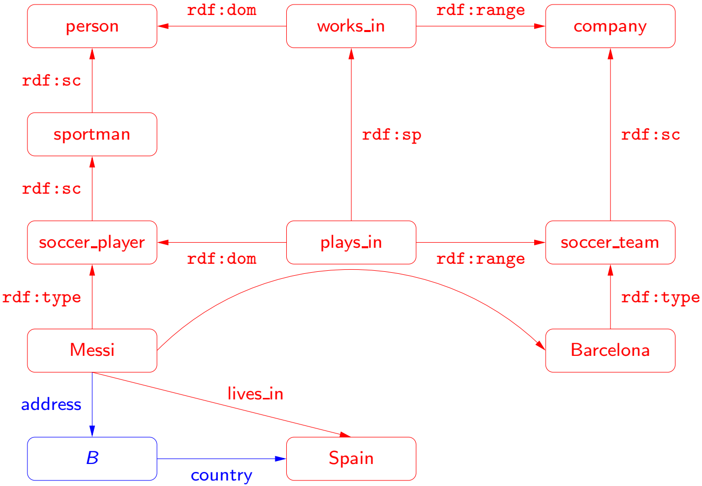
Previous example: A better representation



Previous example: A better representation



Previous example: A better representation

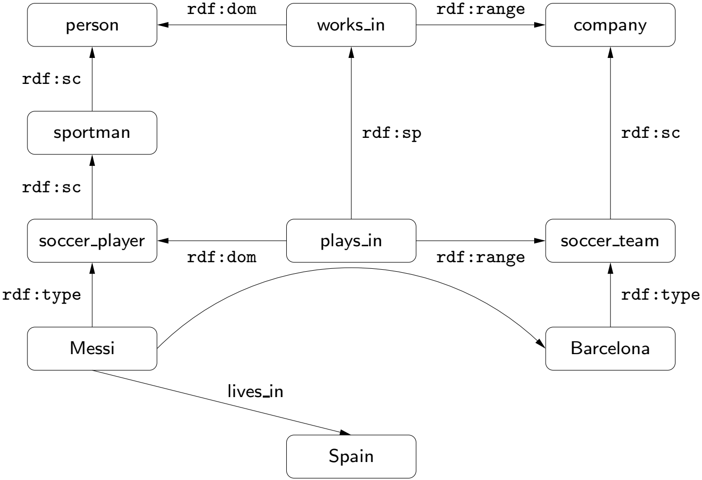


RDF + RDFS

RDFS extends RDF with a schema vocabulary: subPropertyOf (`rdf:sp`), subClassOf (`rdf:sc`), domain (`rdf:dom`), range (`rdf:range`), type (`rdf:type`).

plus *semantics* for this vocabulary

RDFS: Messi is a Person



Semantics of RDFS

Checking whether a triple t is in a graph G is the basic step when reasoning about RDF(S).

- For the case of RDFS, we need to check whether t is implied by G.

Semantics of RDFS

Checking whether a triple t is in a graph G is the basic step when reasoning about RDF(S).

- For the case of RDFS, we need to check whether t is implied by G.

The notion of entailment in RDFS can be defined in terms of classical notions such as model, interpretation, etc.

- As for the case of first-order logic

Semantics of RDFS

Checking whether a triple t is in a graph G is the basic step when reasoning about RDF(S).

- For the case of RDFS, we need to check whether t is implied by G.

The notion of entailment in RDFS can be defined in terms of classical notions such as model, interpretation, etc.

- As for the case of first-order logic

This notion can also be characterized by a set of inference rules.

Semantics of RDFS

Checking whether a triple t is in a graph G is the basic step when reasoning about RDF(S).

- For the case of RDFS, we need to check whether t is implied by G.

The notion of entailment in RDFS can be defined in terms of classical notions such as model, interpretation, etc.

- As for the case of first-order logic

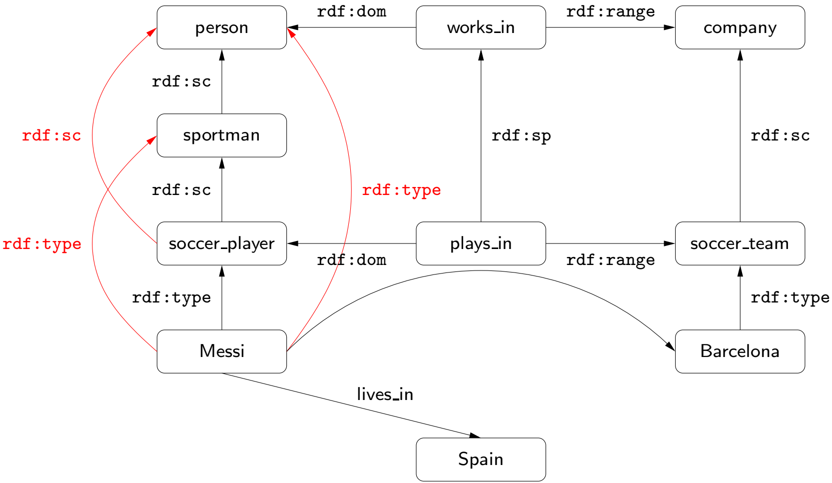
This notion can also be characterized by a set of inference rules.

The closure of an RDFS graph G ($\text{cl}(G)$) is the graph obtained by adding to G all the triples that are implied by G.

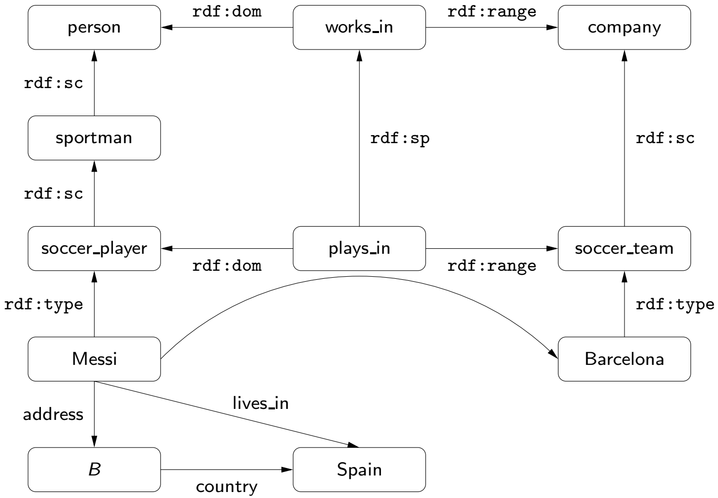
A basic property of the closure:

- G implies t iff $t \in \text{cl}(G)$

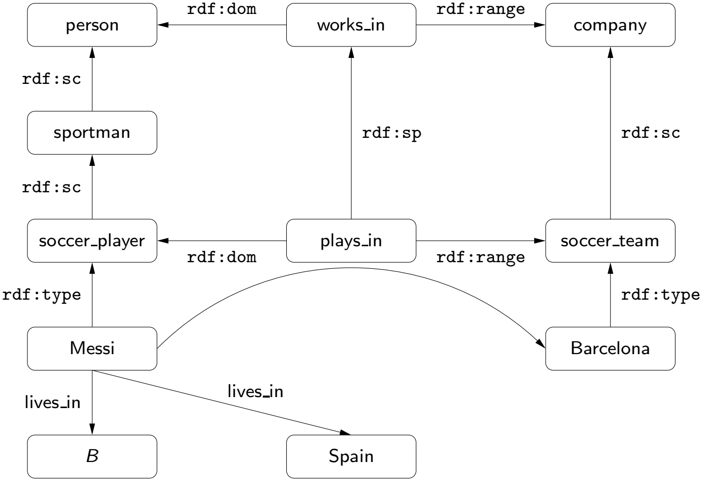
Example: (Messi, rdf:type, person) over the closure



Does the blank node add some information?



What about now?



SPARQL

Querying RDF: SPARQL

- SPARQL is the W3C recommendation query language for RDF (January 2008).
  - SPARQL is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*
- SPARQL is a graph-matching query language.
- A SPARQL query consists of three parts:
  - Pattern matching: optional, union, filtering, ...
  - Solution modifiers: projection, distinct, order, limit, offset, ...
  - Output part: construction of new triples, ...

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{



}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .


}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .

}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

A SPARQL query consists of a:

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator      ?Author .
  ?Paper      dct:partOf      ?Conf .
  ?Conf       swrc:series     conf:iswc .
}
```

A SPARQL query consists of a:

Head: Processing of the variables

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

A SPARQL query consists of a:

Head: Processing of the variables

Body: Pattern matching expression

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC, and their Web pages if this information is available:

```
SELECT ?Author ?WebPage
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .

  OPTIONAL {
      ?Author  foaf:homePage  ?WebPage . }
}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC, and their Web pages if this information is available:

```
SELECT ?Author ?WebPage
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .

  OPTIONAL {
      ?Author  foaf:homePage  ?WebPage . }
}
```

But things can become more complex...

Interesting features of pattern matching on graphs

```
SELECT ?X1 ?X2 ...
  { P1 .
    P2 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2 }



   { P3 .
    P4 }



 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7 }  }

 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns

```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9 }}
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns
- Filtering
- ...
- + several new features in the upcoming version: federation, navigation

```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9
  FILTER ( R ) }}
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns
- Filtering
- ...
- + several new features in the upcoming version: federation, navigation

```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9
  FILTER ( R ) }}
```

What is the (formal) *meaning* of a general SPARQL query?

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

```
{ P1 . P2 }
```

$(\, P_1 \text{ AND } P_2 \,)$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

`?X :name "john"` $(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

`{ P1 . P2 }` $(\, P_1 \text{ AND } P_2 \,)$

`{ P1 OPTIONAL { P2 }}` $(\, P_1 \text{ OPT } P_2 \,)$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

`?X :name "john"` $(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

`{ P1 . P2 }` $(\, P_1 \text{ AND } P_2 \,)$

`{ P1 OPTIONAL { P2 }}` $(\, P_1 \text{ OPT } P_2 \,)$

`{ P1 } UNION { P2 }` $(\, P_1 \text{ UNION } P_2 \,)$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

`{ P1 . P2 }`	$(\, P_1 \text{ AND } P_2 \,)$
`{ P1 OPTIONAL { P2 }}`	$(\, P_1 \text{ OPT } P_2 \,)$
`{ P1 } UNION { P2 }`	$(\, P_1 \text{ UNION } P_2 \,)$
`{ P1 FILTER ( R ) }`	$(\, P_1 \text{ FILTER } R \,)$

A standard algebraic syntax (cont.)

- Explicit precedence/association

Example

```
{ t1
  t2
  OPTIONAL { t3 }
  OPTIONAL { t4 }
  t5
}
```

$$((((t_1 \text{ AND } t_2) \text{ OPT } t_3) \text{ OPT } t_4) \text{ AND } t_5)$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \; \rightarrow \; U \cup L$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \;\rightarrow\; U \cup L$$

Given a mapping μ and a triple pattern t:

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \;\rightarrow\; U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \;\rightarrow\; U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \ V \ \rightarrow \ U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \rightarrow R_1, ?Y \rightarrow R_2, ?Name \rightarrow \text{john}\}$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \; V \; \rightarrow \; U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \rightarrow R_1, ?Y \rightarrow R_2, ?Name \rightarrow \text{john}\}$$

$$t = (?X, \text{ name}, \; ?Name)$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \; V \;\to\; U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \to R_1, ?Y \to R_2, ?Name \to \text{john}\}$$

$$t = (?X, \text{ name}, \; ?Name)$$

$$\mu(t) = (R_1, \text{ name}, \text{ john})$$

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

- dom(μ) is exactly the set of variables occurring in t

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

- $\text{dom}(\mu)$ is exactly the set of variables occurring in t
- $\mu(t) \in G$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

Example

$$G$$

$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(?X,\ \text{name},\ ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

$$[\![(?X, \text{email}, ?E)]\!]_G$$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

$$[\![(?X, \text{email}, ?E)]\!]_G$$

$$\{ \; \mu = \{?X \to R_1, ?E \to \text{J@ed.ex}\} \; \}$$

Example

G

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$
$(R_2, \text{name}, \text{paul})$

$[\![(?X, \text{name}, ?N)]\!]_G$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul

$[\![(?X, \text{email}, ?E)]\!]_G$

	$?X$	$?E$
μ	R_1	J@ed.ex

Example

$$G$$

$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(R_1, \text{webPage}, ?W)]\!]_G$$

$$[\![(R_3, \text{name}, \text{ringo})]\!]_G$$

$$[\![(R_2, \text{name}, \text{paul})]\!]_G$$

Example

$$G$$
$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$[\![(R_1, \text{webPage}, ?W)]\!]_G$

$\{\quad\}$

$[\![(R_3, \text{name}, \text{ringo})]\!]_G$

$[\![(R_2, \text{name}, \text{paul})]\!]_G$

Example

$$G$$

$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(R_1, \text{webPage}, ?W)]\!]_G$$

$$\{\quad\}$$

$$[\![(R_2, \text{name}, \text{paul})]\!]_G$$

$$[\![(R_3, \text{name}, \text{ringo})]\!]_G$$

$$\{\quad\}$$

Example

G

$(R_1,\ \text{name},\ \text{john})$
$(R_1,\ \text{email},\ \text{J@ed.ex})$
$(R_2,\ \text{name},\ \text{paul})$

$[\![(R_1, \text{webPage}, ?W)]\!]_G$

$\{\ \ \}$

$[\![(R_2, \text{name}, \text{paul})]\!]_G$

$\{\ \mu_\emptyset = \{\ \}\ \}$

$[\![(R_3, \text{name}, \text{ringo})]\!]_G$

$\{\ \ \}$

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	john	J@edu.ex	

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	john	J@edu.ex	

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	john	J@edu.ex	
$\mu_1 \cup \mu_3$	R_1	john	P@edu.ex	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	john	J@edu.ex	
$\mu_1 \cup \mu_3$	R_1	john	P@edu.ex	R_2

$\mu_\emptyset = \{ \}$ is compatible with every mapping.

Sets of mappings and operations

Let Ω_1 and Ω_2 be sets of mappings:

Definition

Join: $\Omega_1 \bowtie \Omega_2$

- $\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1, \mu_2 \text{ are compatibles}\}$
- extending mappings in Ω_1 with compatible mappings in Ω_2

will be used to define AND

Sets of mappings and operations

Let Ω_1 and Ω_2 be sets of mappings:

Definition

Join: $\Omega_1 \bowtie \Omega_2$

- $\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2,$ and μ_1, μ_2 are compatibles$\}$
- extending mappings in Ω_1 with compatible mappings in Ω_2

will be used to define AND

Definition

Union: $\Omega_1 \cup \Omega_2$

- $\{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$
- mappings in Ω_1 plus mappings in Ω_2 (the usual set union)

will be used to define UNION

Sets of mappings and operations

Definition

Difference: $\Omega_1 \setminus \Omega_2$

- $\{\mu \in \Omega_1 \mid$ for all $\mu' \in \Omega_2$, μ and μ' are not compatibles$\}$
- mappings in Ω_1 that cannot be extended with mappings in Ω_2

Sets of mappings and operations

Definition

Difference: $\Omega_1 \smallsetminus \Omega_2$

- $\{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2,\ \mu \text{ and } \mu' \text{ are not compatibles}\}$
- mappings in Ω_1 that cannot be extended with mappings in Ω_2

Definition

Left outer join: $\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2)$

- extension of mappings in Ω_1 with compatible mappings in Ω_2
- plus the mappings in Ω_1 that cannot be extended.

will be used to define OPT

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
- $[\![(P_1 \text{ OPT } P_2)]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Example (AND)

$$G: \quad \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

Example (AND)

$$G: \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$ $(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (AND)

G :

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$

$(R_2, \text{name}, \text{paul})$

$(R_3, \text{name}, \text{ringo})$
$(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (AND)

G :

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$

$(R_2, \text{name}, \text{paul})$

$(R_3, \text{name}, \text{ringo})$
$(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (AND)

G :

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$

$(R_2, \text{name}, \text{paul})$

$(R_3, \text{name}, \text{ringo})$
$(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex

Example (OPT)

$$G: \quad \begin{array}{lll} (R_1,\ \text{name},\ \text{john}) & (R_2,\ \text{name},\ \text{paul}) & (R_3,\ \text{name},\ \text{ringo}) \\ (R_1,\ \text{email},\ \text{J@ed.ex}) & & (R_3,\ \text{email},\ \text{R@ed.ex}) \\ & & (R_3,\ \text{webPage},\ \text{www.ringo.com}) \end{array}$$

$$[\![((?X,\ \text{name},\ ?N)\ \text{OPT}\ (?X,\ \text{email},\ ?E))]\!]_G$$

Example (OPT)

$$G: \quad \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G ⟕ [\![(?X, \text{email}, ?E)]\!]_G$$

Example (OPT)

G :

(R_1, name, john)	(R_2, name, paul)	(R_3, name, ringo)
(R_1, email, J@ed.ex)		(R_3, email, R@ed.ex)
		(R_3, webPage, www.ringo.com)

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (OPT)

G :

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$
$(R_2, \text{name}, \text{paul})$
$(R_3, \text{name}, \text{ringo})$
$(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \;⟕\; [\![(?X, \text{email}, ?E)]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (OPT)

G :

$(R_1$, name, john)	$(R_2$, name, paul)	$(R_3$, name, ringo)
$(R_1$, email, J@ed.ex)		$(R_3$, email, R@ed.ex)
		$(R_3$, webPage, www.ringo.com)

$$[\![((?X, \text{name}, ?N) \ \text{OPT} \ (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \ ⟕ \ [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

⟕

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (OPT)

G :

$(R_1, \text{name}, \text{john})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$ $(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G ⟕ [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

⟕

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (OPT)

G :
(R_1, name, john) (R_2, name, paul) (R_3, name, ringo)
(R_1, email, J@ed.ex) (R_3, email, R@ed.ex)
(R_3, webPage, www.ringo.com)

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \;⟕\; [\![(?X, \text{email}, ?E)]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

⟕

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	?X	?N	?E
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (UNION)

$$G: \quad \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

Example (UNION)

$$G: \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

Example (UNION)

G :	$(R_1$, name, john)	$(R_2$, name, paul)	$(R_3$, name, ringo)
	$(R_1$, email, J@ed.ex)		$(R_3$, email, R@ed.ex)
			$(R_3$, webPage, www.ringo.com)

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

Example (UNION)

G : $(R_1, \text{name}, \text{john})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$ $(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

	$?X$	$?Info$
μ_3	R_3	www.ringo.com

Example (UNION)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

$\cup$

	$?X$	$?Info$
μ_3	R_3	www.ringo.com

Example (UNION)

G :

(R_1, name, john)	(R_2, name, paul)	(R_3, name, ringo)
(R_1, email, J@ed.ex)		(R_3, email, R@ed.ex)
		(R_3, webPage, www.ringo.com)

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	?X	?$Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

$\cup$

	?X	?$Info$
μ_3	R_3	www.ringo.com

	?X	?$Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex
μ_3	R_3	www.ringo.com

Boolean filter expressions (value constraints)

In filter expressions we consider

- the equality $=$ among variables and RDF terms
- a unary predicate bound
- boolean combinations ($\wedge$, $\vee$, $\neg$)

A mapping μ satisfies

- $?X = c$ if $\mu(?X) = c$
- $?X = ?Y$ if $\mu(?X) = \mu(?Y)$
- bound($?X$) if μ is defined in $?X$, i.e. $?X \in \mathrm{dom}(\mu)$

Satisfaction of value constraints

- If P is a graph pattern and R is a value constraint then (P FILTER R) is also a graph pattern.

Satisfaction of value constraints

- If P is a graph pattern and R is a value constraint then (P FILTER R) is also a graph pattern.

Definition

Given a graph G

- $[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \text{ satisfies } R\}$
  i.e. mappings in the evaluation of P that satisfy R.

Example (FILTER)

$$G: \quad \begin{array}{lll} (R_1,\ \text{name},\ \text{john}) & (R_2,\ \text{name},\ \text{paul}) & (R_3,\ \text{name},\ \text{ringo}) \\ (R_1,\ \text{email},\ \text{J@ed.ex}) & & (R_3,\ \text{email},\ \text{R@ed.ex}) \\ & & (R_3,\ \text{webPage},\ \text{www.ringo.com}) \end{array}$$

$$[\![((?X,\ \text{name},\ ?N)\ \text{FILTER}\ (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

Example (FILTER)

G :

(R_1, name, john)	(R_2, name, paul)	(R_3, name, ringo)
(R_1, email, J@ed.ex)		(R_3, email, R@ed.ex)
		(R_3, webPage, www.ringo.com)

$$[\![((?X,\ \text{name},\ ?N)\ \text{FILTER}\ (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (FILTER)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ FILTER } (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$?N = \text{ringo} \vee ?N = \text{paul}$

Example (FILTER)

G :

(R_1, name, john)	(R_2, name, paul)	(R_3, name, ringo)
(R_1, email, J@ed.ex)		(R_3, email, R@ed.ex)
		(R_3, webPage, www.ringo.com)

$$[\![((?X,\ \text{name},\ ?N)\ \text{FILTER}\ (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$$?N = \text{ringo} \vee ?N = \text{paul}$$

	$?X$	$?N$
μ_2	R_2	paul
μ_3	R_3	ringo

Example (FILTER)

$$G: \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

Example (FILTER)

$$G: \begin{array}{lll} (R_1, \text{name}, \text{john}) & (R_2, \text{name}, \text{paul}) & (R_3, \text{name}, \text{ringo}) \\ (R_1, \text{email}, \text{J@ed.ex}) & & (R_3, \text{email}, \text{R@ed.ex}) \\ & & (R_3, \text{webPage}, \text{www.ringo.com}) \end{array}$$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg \text{bound}(?E)$

Example (FILTER)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg\,\text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg\,\text{bound}(?E)$

	$?X$	$?N$
μ_2	R_2	paul

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$ $(R_3, \text{email}, \text{R@ed.ex})$
$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg\,\text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg\,\text{bound}(?E)$

	$?X$	$?N$
μ_2	R_2	paul

(a non-monotonic query)

Why do we need/want to formalize SPARQL

A formalization is beneficial

- clarifying corner cases
- helping in the implementation process
- providing solid foundations (we can actually prove properties!)

SELECT (a.k.a. projection)

Besides graph patterns, SPARQL 1.0 allow result forms
the most simple is SELECT

Definition

A SELECT query is an expression

$$(\text{SELECT}\ \ W\ P)$$

where P is a graph pattern and W is a set of variables, or $*$

SELECT (a.k.a. projection)

Besides graph patterns, SPARQL 1.0 allow result forms
the most simple is SELECT

Definition
A SELECT query is an expression

$$(\text{SELECT } W\ P)$$

where P is a graph pattern and W is a set of variables, or $*$

The evaluation of a SELECT query against G is

- $[\![(\text{SELECT } W\ P)]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$
  where $\mu_{|W}$ is the restriction of μ to domain W.
- $[\![(\text{SELECT } *\ P)]\!]_G = [\![P]\!]_G$

Example (SELECT)

$$G : \begin{array}{lll} (R_1,\ \text{name},\ \text{john}) & (R_2,\ \text{name},\ \text{paul}) & (R_3,\ \text{name},\ \text{ringo}) \\ (R_1,\ \text{email},\ \text{J@ed.ex}) & & (R_3,\ \text{email},\ \text{R@ed.ex}) \\ & & (R_3,\ \text{webPage},\ \text{www.ringo.com}) \end{array}$$

$$[\![(\text{SELECT } \{?N, ?E\}\ ((?X,\ \text{name},\ ?N)\ \text{AND}\ (?X,\ \text{email},\ ?E)))]\!]_G$$

Example (SELECT)

G :	$(R_1,$ name, john$)$	$(R_2,$ name, paul$)$	$(R_3,$ name, ringo$)$
	$(R_1,$ email, J@ed.ex$)$		$(R_3,$ email, R@ed.ex$)$
			$(R_3,$ webPage, www.ringo.com$)$

$$[\![(\text{SELECT } \{?N, ?E\} \ ((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E)))]\!]_G$$

SELECT$\{?N, ?E\}$

	$?X$	$?N$	$?E$
μ_1	R_1	john	J@ed.ex
μ_2	R_3	ringo	R@ed.ex

Example (SELECT)

G :

$(R_1,$ name, john$)$	$(R_2,$ name, paul$)$	$(R_3,$ name, ringo$)$
$(R_1,$ email, J@ed.ex$)$		$(R_3,$ email, R@ed.ex$)$
		$(R_3,$ webPage, www.ringo.com$)$

$$[\![(\text{SELECT } \{?N, ?E\} \ ((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E)))]\!]_G$$

SELECT$\{?N, ?E\}$

	$?X$	$?N$	$?E$
μ_1	R_1	john	J@ed.ex
μ_2	R_3	ringo	R@ed.ex

	$?N$	$?E$
$\mu_1\vert_{\{?N,?E\}}$	john	J@ed.ex
$\mu_2\vert_{\{?N,?E\}}$	ringo	R@ed.ex

SPARQL 1.1 introduces several new features

In SPARQL 1.1:

- (SELECT W P) can be used as any other graph pattern
  $\Rightarrow$ sub-queries
- Aggregations via ORDER-BY plus COUNT, SUM, etc.
- Most interesting features: Federation and Navigation

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B reachable from A by a sequence of connections?

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B reachable from A by a sequence of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

SPARQL 1.0 has very limited navigational capabilities

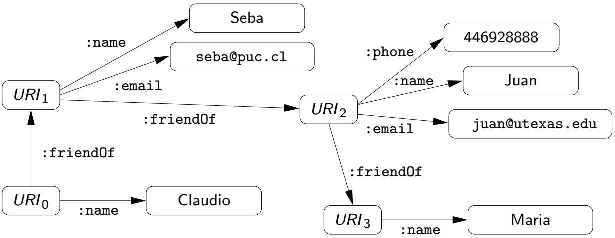
Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

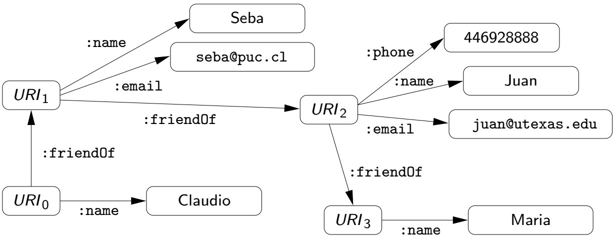
query: is city B reachable from A by a sequence of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

SPARQL 1.1 provides an alternative way for navigating

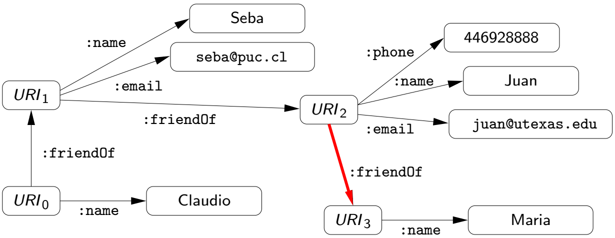


SPARQL 1.1 provides an alternative way for navigating



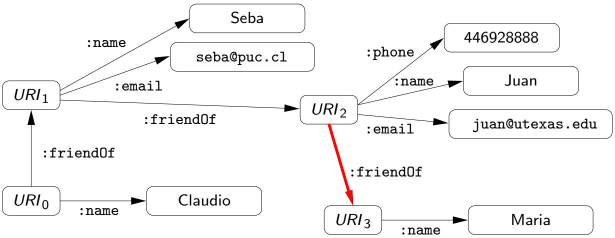

```
SELECT ?X
WHERE
{
   ?X  :friendOf   ?Y .
   ?Y  :name    "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating



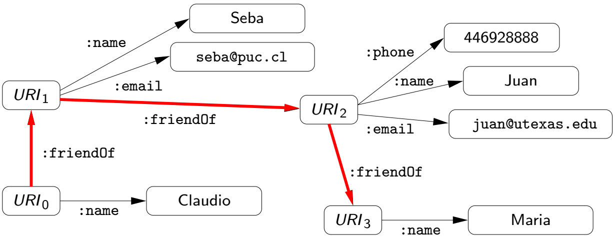

```
SELECT ?X
WHERE
{
   ?X  :friendOf   ?Y .
   ?Y  :name    "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating



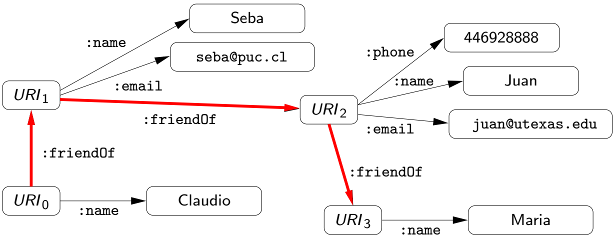

```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating




```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

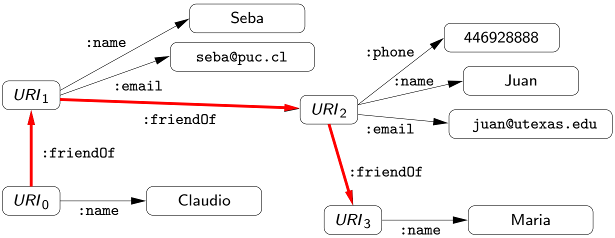
SPARQL 1.1 provides an alternative way for navigating




```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

← *SPARQL 1.1 property path*

SPARQL 1.1 provides an alternative way for navigating




```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

← *SPARQL 1.1 property path*

Idea: navigate RDF graphs using regular expressions

General navigation using regular expressions

Regular expressions define sets of strings using

- concatenation: /
- disjunction: |
- Kleene star: *

Example

Consider strings composed of symbols `a` and `b`

`a/(b)*/a`

defines strings of the form `abbb` $\cdots$ `bbba`.

General navigation using regular expressions

Regular expressions define sets of strings using

- concatenation: /
- disjunction: |
- Kleene star: *

Example

Consider strings composed of symbols `a` and `b`

```
a/(b)*/a
```

defines strings of the form `abbb···bbba`.

Idea: use regular expressions to define paths

- a path `p` satisfies a regular expression `r` if the string composed of the sequence of edges of `p` satisfies expression `r`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

ancestors of `John`

`{ John (:father|:mother)* ?X }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  { `John (:father|:mother)* ?X` }

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  { `?X` `Paris` }

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*                Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus                    Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Mixing regular expressions and SPARQL operators
gives interesting expressive power:

Persons in my professional network that attended the same school

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Mixing regular expressions and SPARQL operators
gives interesting expressive power:

Persons in my professional network that attended the same school

```
{ ?X (:conn)* ?Y .
  ?X (:conn)* ?Z .
  ?Y :sameSchool ?Z }
```

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation
- Mid 2012: semantics change to overcome the raised issues

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation
- Mid 2012: semantics change to overcome the raised issues

The following experimental study is based on [ACP12].