SPARQL and the Rules Layer

Axel Polleres[1]

[1]DERI Galway, National University of Ireland, Galway
axel.polleres@deri.org

European Semantic Web Conference 2007

Outline

The SW Rules layer in a nutshell
  Rules for the Semantic Web

Translating SPARQL to LP style rules languages
  Basic Graph Patterns
  GRAPH Patterns
  UNION Patterns
  OPTIONAL and Negation as failure

Other Rules languages and formats
  SWI Prolog, TRIPLE, N3
  SPARQL and RIF

Outline

The SW Rules layer in a nutshell
- Rules for the Semantic Web

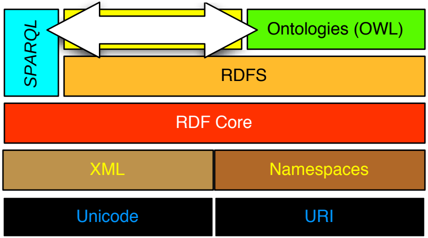
Translating SPARQL to LP style rules languages
- Basic Graph Patterns
- GRAPH Patterns
- UNION Patterns
- OPTIONAL and Negation as failure

Other Rules languages and formats
- SWI Prolog, TRIPLE, N3
- SPARQL and RIF

Back to the layer cake...

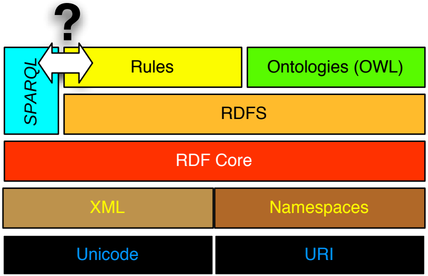
Hope you enjoyed the coffee break. . .

Back to the layer cake...



Bijan will talk about this one in the last part . . .

Back to the layer cake...



. . . Now what about that one?

Rules for/on the Web: Where are we?

- Several proposals for systems and rules languages on the Web usable on top of RDF/RDFS:
  - TRIPLE [Decker et al., 2005]
  - N3 [Berners-Lee et al., 2005]
  - dlvhex [Eiter et al., 2005]
  - SWI-Prolog's semweb library [Wielemaker,]
  - SWRL [Horrocks et al., 2004]
  - SWSL Rules [Battle et al., 2005]
  - WRL, WSML [Angele et al., 2005, de Bruijn et al., 2005]
- RIF working group chartered in Dec 2005 to provide common interchange format (sic! Not a rule language) for the Web:
  - Is currently producing first concrete results and first draft format, in the future likely a common format for the approaches above
  - apart from deductive rules also concerned with other "rules": business rules, ECA rules, (integrity) constraints

Rules for/on the Web: Where are we?

- Several proposals for systems and rules languages on the Web usable on top of RDF/RDFS:
  - TRIPLE [Decker et al., 2005]
  - N3 [Berners-Lee et al., 2005]
  - dlvhex [Eiter et al., 2005]
  - SWI-Prolog's semweb library [Wielemaker,]
  - SWRL [Horrocks et al., 2004]
  - SWSL Rules [Battle et al., 2005]
  - WRL, WSML [Angele et al., 2005, de Bruijn et al., 2005]
- RIF working group chartered in Dec 2005 to provide common interchange format (sic! Not a rule language) for the Web:
  - Is currently producing first concrete results and first draft format, in the future likely a common format for the approaches above
  - apart from deductive rules also concerned with other "rules": business rules, ECA rules, (integrity) constraints

Outline

The SW Rules layer in a nutshell
  - Rules for the Semantic Web

Translating SPARQL to LP style rules languages
  - Basic Graph Patterns
  - GRAPH Patterns
  - UNION Patterns
  - OPTIONAL and Negation as failure

Other Rules languages and formats
  - SWI Prolog, TRIPLE, N3
  - SPARQL and RIF

SPARQL and LP 1/2

- Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^{\text{not}}$)

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

  ```
  SELECT name FROM myAddr WHERE City = "Innsbruck"
    UNION
  SELECT name FROM yourAddresses
  ```

  ```
  answer1(Name) :- myAddr(Name, Street, "Innsbruck", Tel).
  answer1(Name) :- yourAddr(Name, Address).
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble, see Unit 4!

SPARQL and LP 1/2

- Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^{\texttt{not}}$)

  Example: Two tables containing adressbooks

  ```
  myAddr(Name, Street, City, Telephone)
  yourAddr(Name, Address)
  ```

  ```
  SELECT name FROM myAddr WHERE City = "Innsbruck"
    UNION
  SELECT name FROM yourAddresses
  ```

  ```
  answer1(Name) :- myAddr(Name, Street, "Innsbruck", Tel).
  answer1(Name) :- yourAddr(Name, Address).
  ```

  ```
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble, see Unit 4!

SPARQL and LP 1/2

- Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^{\text{not}}$)

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

  ```
  SELECT name FROM myAddr WHERE City = "Innsbruck"
    UNION
  SELECT name FROM yourAddresses
  ```

  ```
  answer1(Name) :- myAddr(Name, Street, "Innsbruck", Tel).
  answer1(Name) :- yourAddr(Name, Address).
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble, see Unit 4!

SPARQL and LP 1/2

- Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^{\text{not}}$)

  Example: Two tables containing adressbooks

  ```
  myAddr(Name, Street, City, Telephone)
  yourAddr(Name, Address)
  ```

  ```
  SELECT name FROM myAddr WHERE City = "Innsbruck"
    UNION
  SELECT name FROM yourAddresses
  ```

  ```
  answer1(Name) :- myAddr(Name, Street, "Innsbruck", Tel).
  answer1(Name) :- yourAddr(Name, Address).
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble, see Unit 4!

SPARQL and LP 1/2

- Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^{\text{not}}$)

  Example: Two tables containing adressbooks

  ```
  myAddr(Name, Street, City, Telephone)
  yourAddr(Name, Address)
  ```

  ```
  SELECT name FROM myAddr WHERE City = "Innsbruck"
    UNION
  SELECT name FROM yourAddresses
  ```

  ```
  answer1(Name) :- myAddr(Name, Street, "Innsbruck", Tel).
  answer1(Name) :- yourAddr(Name, Address).
  ```

  ```
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble, see Unit 4!

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf [URL] (S,P,O)` to import RDF data.

- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf [URL] (S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://con.fusion.at/dlvhex/`):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf [URL] (S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (`http://www.dlvsystem.com/`)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates).
- `rdf [URL] (S,P,O)` is one such built-in to import RDF data.

The example translations in this Unit work similarly using e.g. SWI-Prolog's rdf_db module (see, `http://www.swi-prolog.org/packages/semweb.html`).

SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).
```

```
?- answer1(X,Y,def).
```

SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).
```

```
?- answer1(X,Y,def).
```

SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

“select persons and their names”

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).
```

```
?- answer1(X,Y,def).
```

SPARQL and LP: GRAPH Patterns and NAMED graphs

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
         GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: GRAPH Patterns and NAMED graphs

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
         GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: GRAPH Patterns and NAMED graphs

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
         GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: GRAPH Patterns and NAMED graphs

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
        GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: GRAPH Patterns and NAMED graphs

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
         GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: GRAPH Patterns and NAMED graphs

“select creators of graphs and the persons they know”

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
         GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

SPARQL and LP: UNION Patterns 1/2

UNIONs are split of into several rules:

"select Persons and their names **or** *nicknames"*

```
SELECT ?X ?Y
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Y .} }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,def) :- triple(X,"foaf:name",Y,def).
answer1(X,Y,def) :- triple(X,"foaf:nick",Y,def).
```

SPARQL and LP: UNION Patterns 1/2

UNIONs are split of into several rules:

"select Persons and their names **or** *nicknames"*

```
SELECT ?X ?Y
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Y .} }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,def) :- triple(X,"foaf:name",Y,def).
answer1(X,Y,def) :- triple(X,"foaf:nick",Y,def).
```

SPARQL and LP: UNION Patterns 1/2

UNIONs are split of into several rules:

"select Persons and their names **or** *nicknames"*

```
SELECT ?X ?Y
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Y .} }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,def) :- triple(X,"foaf:name",Y,def).
answer1(X,Y,def) :- triple(X,"foaf:nick",Y,def).
```

SPARQL and LP: UNION Patterns 2/2

What if variables of the of constituent patterns don't coincide?
Slightly different than in SQL!
We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Z .} }
```

Data:
`<alice.org#me> foaf:name "Alice".`
`<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".`
Result:

?X	?Y	?Z
<alice.org#me>	"Alice"	
<ex.org/bob#me>	"Bob"	
<ex.org/bob#me>		"Bobby"

SPARQL and LP: UNION Patterns 2/2

What if variables of the of constituent patterns don't coincide?

Slightly different than in SQL!

We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
          UNION { ?X foaf:nick ?Z .} }
```

Data:

```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:

?X	?Y	?Z
<alice.org#me>	"Alice"	
<ex.org/bob#me>	"Bob"	
<ex.org/bob#me>		"Bobby"

SPARQL and LP: UNION Patterns 2/2

What if variables of the of constituent patterns don't coincide?

Slightly different than in SQL!

We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Z .} }
```

Data:

`<alice.org#me> foaf:name "Alice".`

`<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".`

Result:

?X	?Y	?Z
`<alice.org#me>`	"Alice"	`null`
`<ex.org/bob#me>`	"Bob"	`null`
`<ex.org/bob#me>`	`null`	"Bobby"

SPARQL and LP: UNION Patterns 2/2

What if variables of the of constituent patterns don't coincide?
Slightly different than in SQL!
We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Z .} }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,null,def) :- triple(X,"foaf:name",Y,def).
answer1(X,null,Z,def) :- triple(X,"foaf:nick",Z,def).
```

SPARQL and LP: UNION Patterns 2/2

What if variables of the of constituent patterns don't coincide?
Slightly different than in SQL!
We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
         UNION { ?X foaf:nick ?Z .} }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,null,def) :- triple(X,"foaf:name",Y,def).
answer1(X,null,Z,def) :- triple(X,"foaf:nick",Z,def).
```

SPARQL and LP: *OPTIONAL* Patterns 1/2

"select all persons and optionally their names"

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

OPTIONAL is similar to an OUTER JOIN in SQL, actually it is a combination of a **join** and **set difference**:

$\{P_1\ \texttt{OPTIONAL}\ \{P_2\}\}$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where M_1 and M_2 are variable binding for P_1 and P_2, resp.

SPARQL and LP: *OPTIONAL* Patterns 1/2

“select all persons and optionally their names”

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

OPTIONAL is similar to an OUTER JOIN in SQL, actually it is a combination of a **join** and **set difference**:

$\{P_1 \texttt{ OPTIONAL } \{P_2\}\}: \quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where M_1 and M_2 are variable binding for P_1 and P_2, resp.

SPARQL and LP: *OPTIONAL* Patterns 1/2

"select all persons and optionally their names"

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

OPTIONAL is similar to an OUTER JOIN in SQL, actually it is a combination of a **join** and **set difference**:

$\{P_1 \texttt{ OPTIONAL } \{P_2\}\}$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where M_1 and M_2 are variable binding for P_1 and P_2, resp.

SPARQL's OPTIONAL has "negation as failure", hidden:

- Observation: SPARQL allows to express set difference / negation as failure by combining OPT and !bound

"select all persons *without* an email address"

```
SELECT ?X
WHERE
{
  ?X a ?Person
  OPTIONAL {?X :email ?Email }
  FILTER ( !bound( ?Email ) )
}
```

- Same effect as "NOT EXISTS" in SQL, set difference!.
- We've seen before that OPTIONAL, has set difference inherent, with the "!bound" we get it back again "purely".

SPARQL's OPTIONAL has "negation as failure", hidden:

- Observation: SPARQL allows to express set difference / negation as failure by combining OPT and !bound

"select all persons *without* an email address"

```
SELECT ?X
WHERE
{
  ?X a ?Person
  OPTIONAL {?X :email ?Email }
  FILTER ( !bound( ?Email ) )
}
```

- Same effect as "NOT EXISTS" in SQL, set difference!.
- We've seen before that OPTIONAL, has set difference inherent, with the "!bound" we get it back again "purely".

SPARQL's OPTIONAL has "negation as failure", hidden:

- Observation: SPARQL allows to express set difference / negation as failure by combining OPT and !bound

"select all persons *without* an email address"

```
SELECT ?X
WHERE
{
  ?X a ?Person
  OPTIONAL {?X :email ?Email }
  FILTER ( !bound( ?Email ) )
}
```

- Same effect as "NOT EXISTS" in SQL, set difference!.
- We've seen before that OPTIONAL, has set difference inherent, with the "!bound" we get it back again "purely".

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                        triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                       not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \smallsetminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                        triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                       not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to "emulate" set difference.

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                        triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                       not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                        triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                       not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                    triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                       not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.

SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: $(P_1 \text{ OPT } P_2)$: $\quad M_1 \mathbin{⟕} M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                        triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                          not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.

SPARQL and LP: OPT Patterns – Example

```
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

  <ex.org/bob> foaf:maker _:a.
  _:a a foaf:Person ; foaf:name "Bob";
            foaf:knows _:b.

  _:b a foaf:Person ; foaf:nick "Alice".
  <alice.org/> foaf:maker _:b
```

```
# Graph: alice.org

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

  alice:me a foaf:Person ; foaf:name "Alice" ;
           foaf:knows _:c.

  _:c  a foaf:Person ; foaf:name "Bob" ;
             foaf:nick "Bobby".
```

```
SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }
```

Result:

?X	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

```
{ answer1("_:a","Bob",def), answer1("_:b",null, def),
  answer1("_:c","Bob",def), answer1("alice.org#me","Alice", def) }
```

SPARQL and LP: OPT Patterns – Example

```
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

  <ex.org/bob> foaf:maker _:a.
  _:a a foaf:Person ; foaf:name "Bob";
          foaf:knows _:b.

  _:b a foaf:Person ; foaf:nick "Alice".
  <alice.org/> foaf:maker _:b
```

```
# Graph: alice.org

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

  alice:me a foaf:Person ; foaf:name "Alice" ;
           foaf:knows _:c.

  _:c  a foaf:Person ; foaf:name "Bob" ;
           foaf:nick "Bobby".
```

```
SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }
```

Result:

?X	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

```
{ answer1("_:a","Bob",def), answer1("_:b",null, def),
  answer1("_:c","Bob",def), answer1("alice.org#me","Alice", def) }
```

SPARQL and LP: OPT Patterns – Example

```
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

  <ex.org/bob> foaf:maker _:a.
  _:a a foaf:Person ; foaf:name "Bob";
            foaf:knows _:b.

  _:b a foaf:Person ; foaf:nick "Alice".
  <alice.org/> foaf:maker _:b
```

```
# Graph: alice.org

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

  alice:me a foaf:Person ; foaf:name "Alice" ;
           foaf:knows _:c.

  _:c  a foaf:Person ; foaf:name "Bob" ;
           foaf:nick "Bobby".
```

```
SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }
```

Result:

?X	?N
_:a	"Bob"
_:b	null
_:c	"Bob"
alice.org#me	"Alice"

```
{ answer1("_:a","Bob",def), answer1("_:b",null, def),
  answer1("_:c","Bob",def), answer1("alice.org#me","Alice", def) }
```

SPARQL and LP: OPT Patterns – Nasty Example

Ask for pairs of persons ?$X1$, ?$X2$ who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

?X1	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

$\bowtie$

?X2	?N
_:a	
_:b	"Alice"
_:c	"Bobby"
alice.org#me	

Now this is strange, as we join over unbound variables.

Remark: this pattern is not well-designed, following Unit 4!

SPARQL and LP: OPT Patterns – Nasty Example

Ask for pairs of persons ?$X1$, ?$X2$ who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

?X1	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

$\bowtie$

?X2	?N
_:a	
_:b	"Alice"
_:c	"Bobby"
alice.org#me	

Now this is strange, as we join over unbound variables.

Remark: this pattern is not well-designed, following Unit 4!

SPARQL and LP: OPT Patterns – Nasty Example

Ask for pairs of persons $?X1$, $?X2$ who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

?X1	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

⋈

?X2	?N
_:a	
_:b	"Alice"
_:c	"Bobby"
alice.org#me	

Now this is strange, as we join over unbound variables.

Remark: this pattern is not well-designed, following Unit 4!

SPARQL and LP: OPT Patterns – With our translation?:

?X1	?N
_:a	"Bob"
_:b	null
_:c	"Bob"
alice.org#me	"Alice"

$\bowtie$

?X2	?N
_:a	null
_:b	"Alice"
_:c	"Bobby"
alice.org#me	null

=

?X1	?N	X2
_:b	null	_:a
_:b	null	alice.org#me
alice.org#me	"Alice"	_:b

What's wrong here? Join over `null` , as if it was a normal constant. Compared with SPARQL's notion of compatibility of mappings, this is too cautious!

SPARQL and LP: OPT Patterns – With our translation?:

?X1	?N
_:a	"Bob"
_:b	null
_:c	"Bob"
alice.org#me	"Alice"

⋈

?X2	?N
_:a	null
_:b	"Alice"
_:c	"Bobby"
alice.org#me	null

=

?X1	?N	X2
_:b	null	_:a
_:b	null	alice.org#me
alice.org#me	"Alice"	_:b

What's wrong here? Join over `null` , as if it was a normal constant.
Compared with SPARQL's notion of compatibility of mappings, this is too cautious!

SPARQL and LP: OPT Patterns – Correct Result:

?X1	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

⋈

?X2	?N
_:a	
_:b	"Alice"
_:c	"Bobby"
alice.org#me	

=

?X1	?N	X2
_:a	"Bob"	_:a
_:a	"Bob"	alice.org#me
_:b		_:a
_:b	"Alice"	_:b
_:b	"Bobby"	_:c
_:b		alice.org#me
_:c	"Bob"	_:a
_:c	"Bob"	alice.org#me
alice.org#me	"Alice"	_:a
alice.org#me	"Alice"	_:b
alice.org#me	"Alice"	alice.org#me

SPARQL defines a very brave way of joins: unbound, i.e. `null` should join with anything!

SPARQL and LP: OPT Patterns – Correct Result:

?X1	?N
_:a	"Bob"
_:b	
_:c	"Bob"
alice.org#me	"Alice"

$\bowtie$

?X2	?N
_:a	
_:b	"Alice"
_:c	"Bobby"
alice.org#me	

$=$

?X1	?N	X2
_:a	"Bob"	_:a
_:a	"Bob"	alice.org#me
_:b		_:a
_:b	"Alice"	_:b
_:b	"Bobby"	_:c
_:b		alice.org#me
_:c	"Bob"	_:a
_:c	"Bob"	alice.org#me
alice.org#me	"Alice"	_:a
alice.org#me	"Alice"	_:b
alice.org#me	"Alice"	alice.org#me

SPARQL defines a very brave way of joins: unbound, i.e. `null` should join with anything!

Semantic variations of SPARQL

We could call these alternatives of treatment of possibly `null`-joining values alternative semantics for SPARQL:

- c-joining: cautiously joining semantics
- b-joining: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let's see to how to fix our translation to logic programs...

Semantic variations of SPARQL

We could call these alternatives of treatment of possibly `null`-joining values alternative semantics for SPARQL:

- c-joining: cautiously joining semantics
- b-joining: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let's see to how to fix our translation to logic programs...

Semantic variations of SPARQL

We could call these alternatives of treatment of possibly `null`-joining values alternative semantics for SPARQL:

- c-joining: cautiously joining semantics
- b-joining: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let's see to how to fix our translation to logic programs...

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)      :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

Here is the problem! Join over a *possibly* `null`*-joining variable*

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)      :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

Here is the problem! Join over a *possibly* `null`*-joining variable*

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)      :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

Here is the problem! Join over a *possibly* `null`*-joining variable*

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)      :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

Here is the problem! Join over a *possibly* `null`-*joining variable*

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)      :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

Here is the problem! Join over a *possibly* `null`*-joining variable*

SPARQL and LP: OPT Patterns – Improved!

How do I emulate b-joining Semantics? Solution:

We need to take care for variables which are joined and possibly unbound, due to the special notion of compatibility in SPARQL

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).
answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(null,X2,def).
answer1(N,X1,X2,def) :- answer2(null,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)     :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

SPARQL and LP: OPT Patterns – Improved!

How do I emulate b-joining Semantics? **Solution**:

We need to take care for variables which are joined and possibly unbound, due to the special notion of compatibility in SPARQL

```
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).
answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(null,X2,def).
answer1(N,X1,X2,def) :- answer2(null,X1,def), answer4(N,X2,def).

answer2(N,   X1,def) :- triple(X1,"a","Person",def),
                        triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                        not answer3(X1,def).
answer3(X1,def)     :- triple(X1,"name",N,def).

answer4(N,   X2,def) :- triple(X2,"a","Person",def),
                        triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                        not answer5(X2,def).
answer5(X2,def)     :- triple(X2,"nick",N,def).
```

SPARQL and LP: OPT Patterns

Attention:

- The "fix" we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.
- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

SPARQL and LP: OPT Patterns

Attention:

- The "fix" we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.
- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

SPARQL and LP: OPT Patterns

Attention:

- The “fix” we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.
- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! “query pushing”

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

From SPARQL to Rules . . . Summary!

- With these ingredients any SPARQL query Q can be translated recursively to a Datalog program P_q with a dedicated predicate $\texttt{answer1}_Q$ which contains exactly the answer substitutions for Q.
- The target language is non-recursive Datalog with neg. as failure
- Non-well-designed combinations of OPTIONAL and UNION are nasty and need special care: **Special treatment for the case where possibly `null` values are joined.**
- Full details of the translation in [Polleres, 2007].
- FILTERS not treated in detail, basically an implementation issue, needs a rules engine with support for external built-ins.
- In order to properly deal with the **multiset**-semantics of SPARQL, UNIONS and projections need special care!
- Interesting might also be the other way around! "query pushing"

Prototype engine implemented and available at:

`http://con.fusion.at/dlvhex/sparql-query-evaluation.php`

Outline

The SW Rules layer in a nutshell
- Rules for the Semantic Web

Translating SPARQL to LP style rules languages
- Basic Graph Patterns
- GRAPH Patterns
- UNION Patterns
- OPTIONAL and Negation as failure

Other Rules languages and formats
- SWI Prolog, TRIPLE, N3
- SPARQL and RIF

Other LP style languages

Similar considerations apply to other rule systems that allow to process RDF data, each of which has some syntactic peculiarities. We exemplify here:

- dlvhex
    - Done! SPARQL-plugin available.
- SWI-Prolog
    - similar... rdf_db module supports rdf/3, rdf/4 predicates, analogous to dlvhex rdf built-in.
- TRIPLE
- N3

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

Basic pattern SPARQL query "emulated" in TRIPLE:

```
@PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X foaf:name ?Y .
          ?X a foaf:Person . }
```

- UNION can be done as before.
- TRIPLE doesn't support negation as failure, thus OPTIONAL not possible.

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

Basic pattern SPARQL query "emulated" in TRIPLE:

```
foaf:= 'http://xmlns.com/foaf/0.1/'.
rdf:= 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
FORALL S,P,O S[P->O] <- S[P->O]@'http://alice.org' OR
                        S[P->O]@'http://ex.org/bob'.
FORALL X,Y answer(X,Y) <- (X[rdf:name->Y] AND
                           X[foaf:type->foaf:person]).
```

- UNION can be done as before.
- TRIPLE doesn't support negation as failure, thus OPTIONAL not possible.

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

GRAPH pattern SPARQL query "emulated" in TRIPLE:

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM NAMED <http://alice.org/bob>
FROM NAMED <http://ex.org/bob>
WHERE { ?G foaf:maker ?X .
          GRAPH ?G { ?X foaf:knows ?Y. } }
```

- UNION can be done as before.
- TRIPLE doesn't support negation as failure, thus OPTIONAL not possible.

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

GRAPH pattern SPARQL query “emulated” in TRIPLE:

```
FORALL S,P,O S[P->O] <- S[P->O]@'http://alice.org'.


FORALL X,Y answer(X,Y) <- (G[foaf:maker->X] AND
                           X[foaf:knows->Y]@G).
```

- UNION can be done as before.
- TRIPLE doesn’t support negation as failure, thus OPTIONAL not possible.

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

- UNION can be done as before.
- TRIPLE doesn't support negation as failure, thus OPTIONAL not possible.
- (Negation as failure under the well-founded semantics seems to be a trivial extension though, since TRIPLE is XSB based.)

TRIPLE

- RDF rules processor on top of XSB Prolog, developed by Michael Sintek, Stefan Decker.
- F-Logic style syntax, i.e. triple `S P O.` viewed as *F-Logic* molecule `S[P->O]`
- Special features: module mechanism.

- UNION can be done as before.
- TRIPLE doesn't support negation as failure, thus OPTIONAL not possible.
- (Negation as failure under the well-founded semantics seems to be a trivial extension though, since TRIPLE is XSB based.)

N3

- RDF rules processor, CWM, implemented in python, developed by Dan Conolly, et al.
- N3 logic syntax, an extension of Turtle syntax.
- Special features: has negation as failure (log:notIncludes).
- Semantics... ? Probably perfect model semantics (i.e. only deals with stratified negation as failure)

Basic pattern SPARQL query “emulated” in N3:

```
@PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X foaf:name ?Y .
         ?X a foaf:Person . }
```

N3

- RDF rules processor, CWM, implemented in python, developed by Dan Conolly, et al.
- N3 logic syntax, an extension of Turtle syntax.
- Special features: has negation as failure (log:notIncludes).
- Semantics... ? Probably perfect model semantics (i.e. only deals with stratified negation as failure)

Basic pattern SPARQL query "emulated" in N3:

```
{ <http://alice.org> log:semantics ?A.
  <http://ex.org/bob> log:semantics ?B.
  (?A ?B) log:conjunction ?C.
  ?C log:supports { ?X foaf:name ?Y . ?X a foaf:Person . }
} log:implies  { myQuery hasAnswer (?X ?Y) . } }
```

Remark: We "encode" answer substitutions in triples here.

N3

- RDF rules processor, CWM, implemented in python, developed by Dan Conolly, et al.
- N3 logic syntax, an extension of Turtle syntax.
- Special features: has negation as failure (log:notIncludes).
- Semantics... ? Probably perfect model semantics (i.e. only deals with stratified negation as failure)

GRAPH pattern SPARQL query “emulated” in N3:

```
{ <http://alice.org> log:semantics ?A.
  ?A log:supports { ?G foaf:maker ?X . }
  ?G log:semantics ?B.
  ?B log:supports { ?X foaf:knows ?Y. }
} log:implies  { myQuery hasAnswer (?X ?Y) . } }
```

N3

- RDF rules processor, CWM, implemented in python, developed by Dan Conolly, et al.
- N3 logic syntax, an extension of Turtle syntax.
- Special features: has negation as failure (log:notIncludes).
- Semantics... ? Probably perfect model semantics (i.e. only deals with stratified negation as failure)

How to "emulate" OPTIONAL patterns in N3:

`log:notIncludes` in N3 is negation as failure!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple “webbish” Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

  If an RDF graph contains triples (`P rdfs:range C`) and (`S P O`) then the triple `O rdf:type C` is entailed.

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

```
CONSTRUCT {?O a ?C . }
WHERE { ?P rdfs:range ?C . ?S ?P ?O . }
```

→ More on that in the next Unit!

SPARQL and RIF

- RIF charter requires rules to deal with RDF data
- It is also written in the RIF charter that RIF should compatible to deal with SPARQL queries to access (external) datasets
- Both not yet addressed in WD1, first step:
  - Simple "webbish" Horn-style rules language (RIF Core)
  - Trouble: Has to address incompatibilities at lower levels... e.g.
    - URIs: Qnames in XML vs. RDF treatment of namespaces
    - compatibility with RDFS, OWL (not fully tackled in SPARQL even)
- Last but not least: SPARQL itself may be viewed as a rules language e.g. take the RDFS entailment rule (rdfs3) from [Hayes, 2004]

```
CONSTRUCT {?O a ?C . }
WHERE { ?P rdfs:range ?C . ?S ?P ?O . }
```

→ More on that in the next Unit!

References I

Angele, J. et al. (2005).
Web rule language (WRL).
W3C Member Submission, available from `http://www.w3.org/Submission/WRL/`.

Battle, S. et al. (2005).
Semantic web services framework (SWSF).
W3C Member Submission, available from `http://www.w3.org/Submission/SWSF/`.

Berners-Lee, T., Connolly, D., Prud'homeaux, E., and Scharf, Y. (2005).
Experience with n3 rules.
In *W3C Workshop on Rule Languages for Interoperability*, Washington, D.C., USA.

de Bruijn, J., Fensel, D., Keller, U., Lausen, M. K. H., Krummenacher, R., Polleres, A., and Predoiu, L. (2005).
Web Service Modeling Language (WSML).
W3C.
Member Submission. Available from `http://www.w3.org/Submission/WSML/`.

Decker, S. et al. (2005).
TRIPLE - an RDF rule language with context and use cases.
In *W3C Workshop on Rule Languages for Interoperability*, Washington, D.C., USA.

Eiter, T., Ianni, G., Polleres, A., and Schindlauer, R. (2006).
Answer set programming for the semantic web.
Tutorial at the European Semantic Web Conference (ESWC), see `http://asptut.gibbi.com/`.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2005).
A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming.
In *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, pages 90–96, Edinburgh, UK.

References II

Hayes, P. (2004).
RDF semantics.
Technical report, W3C.
W3C Recommendation, `http://www.w3.org/TR/rdf-mt/`.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. (2004).
SWRL: A semantic web rule language combining OWL and RuleML.
W3C Member Submission.

Pérez, J., Arenas, M., and Gutierrez, C. (2006).
Semantics and complexity of sparql.
Technical Report DB/0605124, arXiv:cs.

Polleres, A. (2007).
From SPARQL to rules (and back).
In *Proceedings of the 16th World Wide Web Conference (WWW2007)*, Banff, Canada.
Extended technical report version available at
`http://www.polleres.net/publications/GIA-TR-2006-11-28.pdf`.

Wielemaker, J.
SWI-Prolog Semantic Web Library.
available at `http://www.swi-prolog.org/packages/semweb.html`.