From SPARQL to Rules (and back)

Axel Polleres[1]

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

World Wide Web Conference 2007

Outline

Rules and SPARQL

- Rules for the Semantic Web

From SPARQL to (LP style) rules . . .

- Basic Graph Patterns
- GRAPH Patterns
- UNION Patterns
- OPTIONAL and Negation as failure

. . . and back

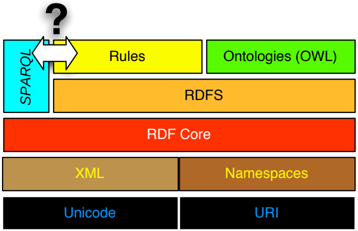
- Use SPARQL as rules
- Mixing data and rules

Rules for/on the Web: Where are we?

- Several existing systems and rules languages on top of RDF/RDFS:
  - TRIPLE , N3/CWM, dlvhex , SWI-Prolog's SW library
- RIF about to make those interoperable by providing a common exchange format
- How to combine SPARQL with (Logic Programming style) rules languages is unclear
- Rule languages are closely related to query languages: Datalog!
- BTW: How do we integrate with RDFS, OWL?

Rules for/on the Web: Where are we?

- Several existing systems and rules languages on top of RDF/RDFS:
  - TRIPLE , N3/CWM, dlvhex , SWI-Prolog's SW library
- RIF about to make those interoperable by providing a common exchange format
- How to combine SPARQL with (Logic Programming style) rules languages is unclear
- Rule languages are closely related to query languages: Datalog!
- BTW: How do we integrate with RDFS, OWL?



Outline

Rules and SPARQL
Rules for the Semantic Web

From SPARQL to (LP style) rules . . .
Basic Graph Patterns
GRAPH Patterns
UNION Patterns
OPTIONAL and Negation as failure

. . . and back
Use SPARQL as rules
Mixing data and rules

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 WHERW City = "Calgary"
  UNION
SELECT name FROM yourAddresses
```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble [Perez et al., 2006]!

SPARQL and LP 1/2

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

Example: Two tables containing adressbooks

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

```
SELECT name FROM myAddr WHERW City = "Calgary"
  UNION
SELECT name FROM yourAddresses
```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble [Perez et al., 2006]!

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 WHERW City = "Calgary"
  UNION
SELECT name FROM yourAddresses
```

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

?- answer1(Name).
```

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble [Perez et al., 2006]!

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 WHERW City = "Calgary"
  UNION
SELECT name FROM yourAddresses
```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble [Perez et al., 2006]!

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 WHERW City = "Calgary"
  UNION
SELECT name FROM yourAddresses
```

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

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION probably cause some trouble [Perez et al., 2006]!

SPARQL and LP 2/2

We start with Datalog with some additional assumptions:

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- We do it by example here, find the formal stuff in the paper!

(**Note:** *The example translations here are based on dlvhex (`http://con.fusion.at/dlvhex/`) syntax, 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 start with Datalog with some additional assumptions:

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- We do it by example here, find the formal stuff in the paper!

(**Note:** *The example translations here are based on dlvhex (`http://con.fusion.at/dlvhex/`) syntax, 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 start with Datalog with some additional assumptions:

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- We do it by example here, find the formal stuff in the paper!

(**Note:** *The example translations here are based on dlvhex (`http://con.fusion.at/dlvhex/`) syntax, 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 start with Datalog with some additional assumptions:

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- We do it by example here, find the formal stuff in the paper!

(**Note:** *The example translations here are based on dlvhex (`http://con.fusion.at/dlvhex/`) syntax, 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

"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).
```

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).
```

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**:

$\{P_1$ `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**:

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

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

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

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

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

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

```
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 from the paper

# Graph: ex.org/bob	# Graph: alice.org
@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	@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 from the paper

```
# 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 from the paper

# Graph: ex.org/bob	# Graph: alice.org

```
# 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 Pérez et al. [Perez et al., 2006]!

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 Pérez et al. [Perez et al., 2006]!

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 Pérez et al. [Perez et al., 2006]!

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 normative semantics is too cautious!

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 normative semantics is too cautious!

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!

?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!

SPARQL and LP: OPT Patterns – third alternative

One could think of a third alternative:

?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
alice.org#me	"Alice"	_:b

In RDBMS implementations of OUTER JOINS, NULL values usually don't join with anything, i.e. this is more strict than the current SPARQL definition!

SPARQL and LP: OPT Patterns – third alternative

One could think of a third alternative:

?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
alice.org#me	"Alice"	_:b

In RDBMS implementations of OUTER JOINS, NULL values usually don't join with anything, i.e. this is more strict than the current SPARQL definition!

Semantic variations of SPARQL

According to these three alternatives of treatment of possibly `null`-joining variables, the paper formally defines three semantics for SPARQL:

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

Which is the most intuitive? Open issue.

Now let's get back to our translation to logic programs...

Semantic variations of SPARQL

According to these three alternatives of treatment of possibly `null`-joining variables, the paper formally defines three semantics for SPARQL:

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

Which is the most intuitive? Open issue.

Now let's get back to our translation to logic programs...

Semantic variations of SPARQL

According to these three alternatives of treatment of possibly `null`-joining variables, the paper formally defines three semantics for SPARQL:

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

Which is the most intuitive? Open issue.

Now let's get back to 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

s-joining semantics can be similarly emulated.

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 [Perez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of the paper [Polleres, 2006]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

SPARQL and LP: OPT Patterns

s-joining semantics can be similarly emulated.

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 [Perez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of the paper [Polleres, 2006]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

SPARQL and LP: OPT Patterns

s-joining semantics can be similarly emulated.

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 [Perez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of the paper [Polleres, 2006]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!

From SPARQL to Rules . . .

- 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 the paper
- FILTERS not treated in detail, basically an implementation issue, needs special built-ins.

From SPARQL to Rules . . .

- 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 the paper
- FILTERS not treated in detail, basically an implementation issue, needs special built-ins.

From SPARQL to Rules . . .

- 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 the paper
- FILTERS not treated in detail, basically an implementation issue, needs special built-ins.

From SPARQL to Rules . . .

- 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 the paper
- FILTERS not treated in detail, basically an implementation issue, needs special built-ins.

From SPARQL to Rules . . .

- 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 the paper
- FILTERS not treated in detail, basically an implementation issue, needs special built-ins.

. . . and back

Some more things discussed in the paper (appetizer):

- Extend the translation to cover CONSTRUCT queries
- CONSTRUCTs themselves can be viewed as rules! Our translation sets the basis for querying combined sets of RDF data and CONSTRUCT queries! (thus the "and back")!
- The translation can serve as a basis for extensions of SPARQL, e.g. nested queries (currently working on implementing these)
- The translation can be easily combined with translations for RDFS, OWL fragments (e.g. ter Horst's fragment [ter Horst, 2005]) $\Rightarrow$ extended entailment regimes for SPARQL!

Some more things discussed in the paper (appetizer):

- Extend the translation to cover CONSTRUCT queries
- CONSTRUCTs themselves can be viewed as rules! Our translation sets the basis for querying combined sets of RDF data and CONSTRUCT queries! (thus the "and back")!
- The translation can serve as a basis for extensions of SPARQL, e.g. nested queries (currently working on implementing these)
- The translation can be easily combined with translations for RDFS, OWL fragments (e.g. ter Horst's fragment [ter Horst, 2005]) $\Rightarrow$ extended entailment regimes for SPARQL!

. . . and back

Some more things discussed in the paper (appetizer):

- Extend the translation to cover CONSTRUCT queries
- CONSTRUCTs themselves can be viewed as rules! Our translation sets the basis for querying combined sets of RDF data and CONSTRUCT queries! (thus the "and back")!
- The translation can serve as a basis for extensions of SPARQL, e.g. nested queries (currently working on implementing these)
- The translation can be easily combined with translations for RDFS, OWL fragments (e.g. ter Horst's fragment [ter Horst, 2005]) $\Rightarrow$ extended entailment regimes for SPARQL!

. . . and back

Some more things discussed in the paper (appetizer):

- Extend the translation to cover CONSTRUCT queries
- CONSTRUCTs themselves can be viewed as rules! Our translation sets the basis for querying combined sets of RDF data and CONSTRUCT queries! (thus the "and back")!
- The translation can serve as a basis for extensions of SPARQL, e.g. nested queries (currently working on implementing these)
- The translation can be easily combined with translations for RDFS, OWL fragments (e.g. ter Horst's fragment [ter Horst, 2005]) $\Rightarrow$ extended entailment regimes for SPARQL!

. . . and back

Some more things discussed in the paper (appetizer):

- Extend the translation to cover CONSTRUCT queries
- CONSTRUCTs themselves can be viewed as rules! Our translation sets the basis for querying combined sets of RDF data and CONSTRUCT queries! (thus the "and back")!
- The translation can serve as a basis for extensions of SPARQL, e.g. nested queries (currently working on implementing these)
- The translation can be easily combined with translations for RDFS, OWL fragments (e.g. ter Horst's fragment [ter Horst, 2005]) $\Rightarrow$ extended entailment regimes for SPARQL!

CONSTRUCT 1/3

CONSTRUCTs themselves may be viewed as rules over RDF themselves.
How to handle CONSTRUCT in the outlined translation to LP?

```
CONSTRUCT  ?X foaf:name ?Y . ?X a foaf:Person .
WHERE { ?X vCard:FN ?Y }.
```

For blanknode-free CONSTRUCTs our translation can be simply extended:

```
triple(X,foaf:name,Y,constructed) :-
            triple(X,rdf:type,foaf:Person,default).
```

and export the RDF triples from predicate

```
triple(S,P,O,constructed)
```

in post-processing to get the constructed RDF graph

CONSTRUCT 1/3

CONSTRUCTs themselves may be viewed as rules over RDF themselves.
How to handle CONSTRUCT in the outlined translation to LP?

```
CONSTRUCT  ?X foaf:name ?Y . ?X a foaf:Person .
WHERE { ?X vCard:FN ?Y }.
```

For blanknode-free CONSTRUCTs our translation can be simply extended:

```
triple(X,foaf:name,Y,constructed) :-
            triple(X,rdf:type,foaf:Person,default).
```

and export the RDF triples from predicate

```
triple(S,P,O,constructed)
```

in post-processing to get the constructed RDF graph

CONSTRUCT 1/3

CONSTRUCTs themselves may be viewed as rules over RDF themselves.
How to handle CONSTRUCT in the outlined translation to LP?

```
CONSTRUCT  ?X foaf:name ?Y . ?X a foaf:Person .
WHERE { ?X vCard:FN ?Y }.
```

For blanknode-free CONSTRUCTs our translation can be simply extended:

```
triple(X,foaf:name,Y,constructed) :-
            triple(X,rdf:type,foaf:Person,default).
```

and export the RDF triples from predicate

```
triple(S,P,O,constructed)
```

in post-processing to get the constructed RDF graph

CONSTRUCT 2/3

More interesting: With this translation, we get for free a way to process mixed RDF and SPARQL CONSTRUCTs in ONE file.

Mock-up syntax, mixing TURTLE and SPARQL to describe implicit data within RDF:

foafWithImplicitdData.rdf

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
:me foaf:knows [foaf:name "Marcelo Arenas"],
               [foaf:name "Claudio Gutierrez"],
               [foaf:name "Bijan Parsia"],
               [foaf:name "Jorge Perez"],
               [foaf:name "Andy Seaborne"].
CONSTRUCT{ :me foaf:knows ?X }
FROM <http://www.deri.ie/about/team>
WHERE { ?X a foaf:Person. }
```

CONSTRUCT 2/3

More interesting: With this translation, we get for free a way to process mixed RDF and SPARQL CONSTRUCTs in ONE file.

Mock-up syntax, mixing TURTLE and SPARQL to describe implicit data within RDF:

foafWithImplicitdData.rdf

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
:me foaf:knows [foaf:name "Marcelo Arenas"],
               [foaf:name "Claudio Gutierrez"],
               [foaf:name "Bijan Parsia"],
               [foaf:name "Jorge Perez"],
               [foaf:name "Andy Seaborne"].
CONSTRUCT{ :me foaf:knows ?X }
FROM <http://www.deri.ie/about/team>
WHERE { ?X a foaf:Person.  }
```

CONSTRUCT 3/3

Attention! If you apply the translation to LP and two RDF+CONSTRUCT files refer mutually to each other, you might get a **recursive** program!

- even non-stratified negation as failure!
- two basic semantics for such "networked RDF graphs" possible:
  - stable [Polleres, 2006]
  - well-founded [Schenk and Staab, 2007]

Outlook

- Prototype implemented and available at http://con.fusion.at/dlvhex/
- Tight integration with existing rules engines possible:
  - Opens up body of optimization work!
  - SPARQL queries in rule bodies
- Most recent working draft of SPARQL has a rel.algebra that slightly deviates from [Perez et al., 2006]:
  - tuple-based instead of set-based
  - FILTERs treated non-local
- Translation can be adapted with minor modifications (personal discussion with editors.)

Thank you! Questions please!

Outlook

- Prototype implemented and available at `http://con.fusion.at/dlvhex/`
- Tight integration with existing rules engines possible:
  - Opens up body of optimization work!
  - SPARQL queries in rule bodies
- Most recent working draft of SPARQL has a rel.algebra that slightly deviates from [Perez et al., 2006]:
  - tuple-based instead of set-based
  - FILTERs treated non-local
- Translation can be adapted with minor modifications (personal discussion with editors.)

Thank you! Questions please!

Outlook

- Prototype implemented and available at
  http://con.fusion.at/dlvhex/
- Tight integration with existing rules engines possible:
  - Opens up body of optimization work!
  - SPARQL queries in rule bodies
- Most recent working draft of SPARQL has a rel.algebra that slightly deviates from [Perez et al., 2006]:
  - tuple-based instead of set-based
  - FILTERs treated non-local
- Translation can be adapted with minor modifications (personal discussion with editors.)

Thank you! Questions please!

Outlook

- Prototype implemented and available at http://con.fusion.at/dlvhex/
- Tight integration with existing rules engines possible:
  - Opens up body of optimization work!
  - SPARQL queries in rule bodies
- Most recent working draft of SPARQL has a rel.algebra that slightly deviates from [Perez et al., 2006]:
  - tuple-based instead of set-based
  - FILTERs treated non-local
- Translation can be adapted with minor modifications (personal discussion with editors.)

Thank you! Questions please!

Outlook

- Prototype implemented and available at `http://con.fusion.at/dlvhex/`
- Tight integration with existing rules engines possible:
  - Opens up body of optimization work!
  - SPARQL queries in rule bodies
- Most recent working draft of SPARQL has a rel.algebra that slightly deviates from [Perez et al., 2006]:
  - tuple-based instead of set-based
  - FILTERs treated non-local
- Translation can be adapted with minor modifications (personal discussion with editors.)

Thank you! Questions please!

References

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

Polleres, A. (2006).
SPARQL Rules!
Technical Report GIA-TR-2006-11-28, Universidad Rey Juan Carlos.

Schenk, S. and Staab, S. (2007).
Networked rdf graph networked rdf graphs.
Technical Report 3/2007, Universsity of Koblenz.
available at `http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf`.

ter Horst, H. J. (2005).
Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary.
Journal of Web Semantics, 3(2).