Next Steps on SPARQL

Axel Polleres (DERI Galway)

Joint work with:
R. Schindlauer (Univ Calabria/TU Vienna), G. Frazzingaro (Univ Calabria), T.Krennwallner (DERI Galway/TU Vienna), F. Scharffe (LFU Innsbruck)

January 09, 2008

Outline

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 = "Bolzano"
    UNION
  SELECT name FROM yourAddresses
  ```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs

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 = "Bolzano"
    UNION
  SELECT name FROM yourAddresses
  ```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs

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 = "Bolzano"
    UNION
  SELECT name FROM yourAddresses
  ```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs

SPARQL and LP 1/2

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

  Example: Two tables containing adressbooks

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

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

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs

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 = "Bolzano"
    UNION
  SELECT name FROM yourAddresses
  ```

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

- That was easy... Now what about SPARQL?
- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://www.kr.tuwien.ac.at/research/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, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://www.kr.tuwien.ac.at/research/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, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://www.kr.tuwien.ac.at/research/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, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://www.kr.tuwien.ac.at/research/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, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.

SPARQL and LP 2/2

We take as an example the language of dlvhex (`http://www.kr.tuwien.ac.at/research/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, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.

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: FILTERs

FILTERs are used to filter the result set of a query.
FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M . ?X :age ?Age .
          FILTER( ?Age > 30 )
      }
```

```
answer1(X,def) :-
   triple(X,foaf:mbox,M,def), triple(X,:age,Age,def),
   Age > 30.
```

unbound variables in FILTERs need to be replaced by constant , to avoid unsafe rules.

SPARQL and LP: FILTERs

FILTERs are used to filter the result set of a query.
FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M . ?X :age ?Age .
          FILTER( ?Age > 30 )
      }
```

```
answer1(X,def) :-
   triple(X,foaf:mbox,M,def), triple(X,:age,Age,def),
   Age > 30.
```

unbound variables in FILTERs need to be replaced by constant , to avoid unsafe rules.

SPARQL and LP: FILTERs

FILTERs are used to filter the result set of a query.
FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M .
        FILTER( ?Age > 30 )
      }
```

```
answer1(X,def) :-
   triple(X,foaf:mbox,M,def),
   null > 30.
```

unbound variables in FILTERs need to be replaced by constant , to avoid unsafe rules.

SPARQL and LP: UNION Patterns 1/2

UNIONs are split off 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 off 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 off 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"	
<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** (see [Pérez et al., 2006]):

$\{P_1$ `OPTIONAL` $\{P_2\}\}$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \smallsetminus \mathcal{M}_2)$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are variable bindings 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** (see [Pérez et al., 2006]):

$\{P_1 \texttt{ OPTIONAL } \{P_2\}\}$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are variable bindings 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** (see [Pérez et al., 2006]):

$\{P_1 \texttt{ OPTIONAL } \{P_2\}\}$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are variable bindings 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 FILTER !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 FILTER !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 FILTER !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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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: $(P_1 \text{ OPT } P_2)$: $\mathcal{M}_1 ⟕ \mathcal{M}_2 = (\mathcal{M}_1 \bowtie \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)$

```
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 [Pérez 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., 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., 2006]

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 – 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 – 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"

⋈

?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!

Outline

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
    1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
    2. SPARQL defines a multi-set semantics.
    3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
    4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification compliance

That's all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)

SPARQL Specification: ORDER BY, LIMIT, OFFSET

- Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:

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

```
SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1
```

Result: `{ answer1("Bob",def), answer1("Alice",def) }`

Sort answer set by parameter corresponding to `?Y` (ORDER BY), only output first result (LIMIT 1) ⇒ `"Alice"`

SPARQL Specification: ORDER BY, LIMIT, OFFSET

- Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:

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

```
SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1
```

Result: `{ answer1("Bob",def), answer1("Alice",def) }`
Sort answer set by parameter corresponding to `?Y` (ORDER BY),
only output first result (LIMIT 1) ⇒ `"Alice"`

SPARQL Specification: ORDER BY, LIMIT, OFFSET

- Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:

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

```
SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1
```

Result: `{ answer1("Bob",def), answer1("Alice",def) }`
Sort answer set by parameter corresponding to `?Y` (ORDER BY), only output first result (LIMIT 1) ⇒ `"Alice"`

SPARQL Specification: ORDER BY, LIMIT, OFFSET

- Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:

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

```
SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1
```

Result: `{ answer1("Bob",def), answer1("Alice",def) }`
Sort answer set by parameter corresponding to `?Y` (ORDER BY), only output first result (LIMIT 1) $\Rightarrow$ `"Alice"`

SPARQL Specification: multi-set semantics

1. **be careful with projections (SELECT)**
2. add some machinery for UNIONs

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

```
SELECT ?Y WHERE {?X foaf:name ?Y }
```

```
answer1(Y,def) :- triple(X,foaf:name,Y,def).
```

Answer set: { `answer("Bob")` },
but expected 2 (identical) solutions!

SPARQL Specification: multi-set semantics

1. **be careful with projections (SELECT)**
2. add some machinery for UNIONs

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

```
SELECT ?Y WHERE {?X foaf:name ?Y }
```

```
answer1(Y,def) :- triple(X,foaf:name,Y,def).
```

Answer set: { `answer("Bob")` },
but expected 2 (identical) solutions!

SPARQL Specification: multi-set semantics

1. **be careful with projections (SELECT)**
2. add some machinery for UNIONs

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

```
SELECT ?Y WHERE {?X foaf:name ?Y }
```

```
answer1(Y,def) :- triple(X,foaf:name,Y,def).
```

Answer set: `{ answer("Bob") }`,
but expected 2 (identical) solutions!

SPARQL Specification: multi-set semantics

1. **be careful with projections (SELECT)**
2. add some machinery for UNIONs

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

```
SELECT ?Y WHERE {?X foaf:name ?Y }
```

```
answer1(X,Y,def) :- triple(X,foaf:name,Y,def).
```

Answer set: `{ answer1(...,"Bob"), answer1(...,"Bob") }`,
2 solutions, leave projection to postprocessing !

SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. **add some machinery for UNIONs**

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

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

answer1(?N,?X,def) :- triple(X,foaf:name,Y,def).
answer1(?N,?X,def) :- triple(X,foaf:nick,Y,def).

Answer set: `{ answer1(..., "Bob"), answer1(..., "Bobby"), answer1(..., "Bob") }`,
but expected 4 solutions!

SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. **add some machinery for UNIONs**

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

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

answer1(?N,?X,def) :- triple(X,foaf:name,Y,def).
answer1(?N,?X,def) :- triple(X,foaf:nick,Y,def).

Answer set: `{ answer1(..., "Bob"), answer1(..., "Bobby"), answer1(..., "Bob") }`,
but expected 4 solutions!

SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. **add some machinery for UNIONs**

Data:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

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

answer1(?N,?X,1,def) :- triple(X,foaf:name,Y,def).
answer1(?N,?X,2,def) :- triple(X,foaf:nick,Y,def).

Answer set: `{ answer1(...,"Bob"), answer1(...,"Bobby"), answer1(...,"Bob"), answer1(...,"Bob") }`,
Add a new constant for any "branch" of a UNION.

SPARQL Specification: FILTER expressions in OPTIONAL patterns

“select names and email addresses only of those older than 30”

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
                     OPTIONAL { ?X foaf:mbox ?M . FILTER(?Age > 30) }}
```

Needs 3 case distinctions:

- There is an email address and the FILTER is fulfilled (join)
- There is an email address and the FILTER is not fulfilled (leave ?M unbound)
- There is no email address (leave ?M unbound)

SPARQL Specification: FILTER expressions in OPTIONAL patterns

"select names and email addresses only of those older than 30"

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
                     OPTIONAL { ?X foaf:mbox ?M . FILTER(?Age > 30) }}
```

Needs 3 case distinctions:

- There is an email address and the FILTER is fulfilled (join)
- There is an email address and the FILTER is not fulfilled (leave ?M unbound)
- There is no email address (leave ?M unbound)

SPARQL Specification: FILTER expressions in OPTIONAL patterns

"select names and email addresses only of those older than 30"

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
                     OPTIONAL { ?X foaf:mbox ?M . FILTER(?Age > 30) }}
```

```
answer1_P(Age,N,M,X,def) :- triple_Q(X,foaf:name,N,def), triple_Q(X,:age,Age,def),
                            answer2_P(M,X,def), Age > 30.
answer1_P(Age,N,null,X,def) :- triple_Q(X,foaf:name,N,def),
                               triple_Q(X,:age,Age,def),
                               answer2_P(M,X,def), not Age > 30.
answer1_P(Age,N,null,X,def) :- triple_Q(X,foaf:name,N,def),
                               triple_Q(X,:age,Age,def), not answer2'_P(X,def).
answer2_P(M,X,def) :- triple_Q(X,foaf:mbox,M,def).
answer2'_P(X,def) :- answer2_P(M,X,def).
answer_Q(N,M) :- answer1_P(Age,N,M,X,def).
```

Outline

Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to **publish** mappings/alignments? This is important to make *Open Linked Data*[1] happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

[1] Combining RDF data that is "out there", e.g. Sindice, DBPedia, SWPipes etc.

Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to **publish** mappings/alignments? This is important to make *Open Linked Data*[1] happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

[1] Combining RDF data that is "out there", e.g. Sindice, DBPedia, SWPipes etc.

Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to **publish** mappings/alignments? This is important to make *Open Linked Data*[1] happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

[1] Combining RDF data that is "out there", e.g. Sindice, DBPedia, SWPipes etc.

Use Case – Ontology Alignment/Mapping

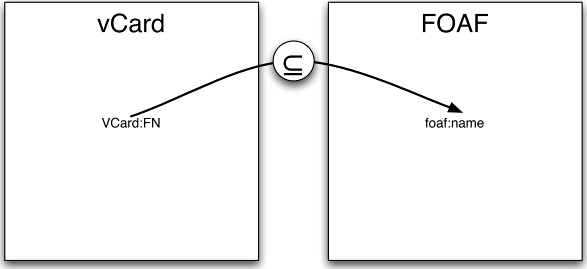
- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to **publish** mappings/alignments? This is important to make *Open Linked Data*[1] happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

[1] Combining RDF data that is "out there", e.g. Sindice, DBPedia, SWPipes etc.

Mapping Scenarios

Map from vCard to FOAF:

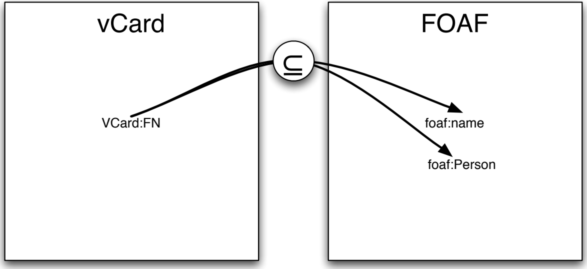


Expressible by `rdfs:subPropertyOf`:

```
VCard:FN rdfs:subPropertyoF foaf:name .
```

Mapping Scenarios

Map from vCard to FOAF:

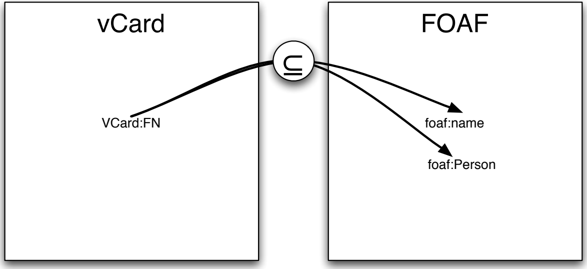


Also expressible in RDFS or in OWL DL:

```
VCard:FN rdfs:subPropertyOf foaf:name.
VCard:FN rdfs:domain foaf:Person.
```

Mapping Scenarios

Map from vCard to FOAF:



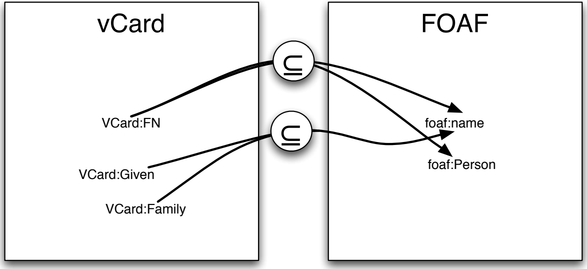
Also expressible in RDFS or in OWL DL:

$\texttt{VCard:FN} \sqsubseteq \texttt{foaf:name}$

$\exists\texttt{VCard:FN}.\top \sqsubseteq \texttt{foaf:Person}$

Mapping Scenarios

Map from vCard to FOAF:



Needs string concatenation, not expressible in OWL or RDFS...

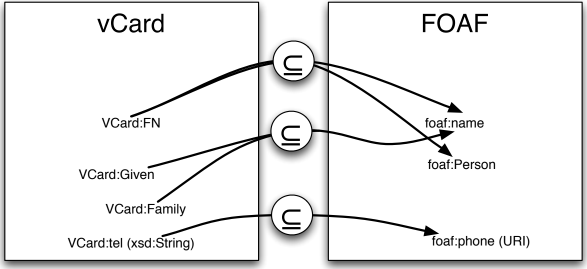
maybe SWRL can help, but

(1) implementations missing

(2) no W3C stamp

Mapping Scenarios

Map from vCard to FOAF:



What shall we do here?
Needs conversion from String to rdf:Resource (URI)...how?
Let's see what SPARQL can do for us...

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS?
(SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS?
(SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS?
(SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS?
(SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)

Mapping by SPARQL

Observation:

SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

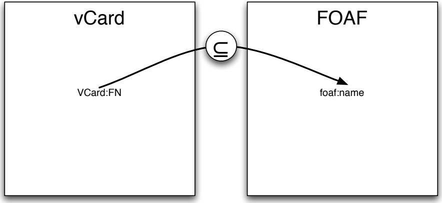
- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)

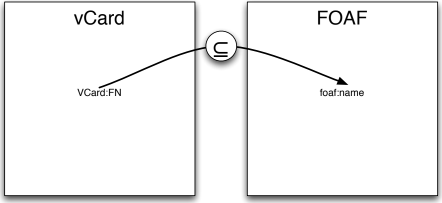
Example 1




```
CONSTRUCT { ?X foaf:name ?Y }
WHERE     { ?X VCard:FN  ?Y }
```

Easy!

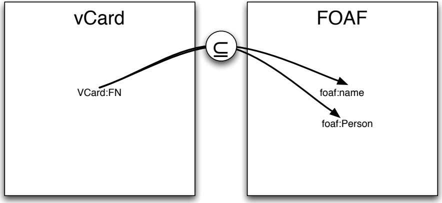
Example 1




```
CONSTRUCT { ?X foaf:name ?Y }
WHERE     { ?X VCard:FN  ?Y }
```

Easy!

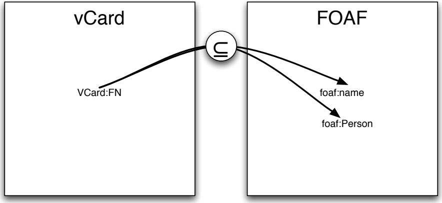
Example 2




```
CONSTRUCT { ?X foaf:name ?Y . ?X rdf:type foaf:person . }
WHERE     { ?X VCard:FN  ?Y }
```

No problem either.

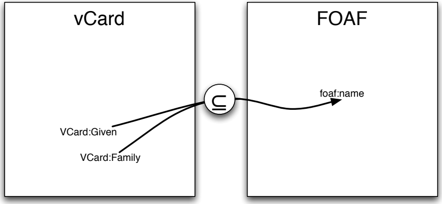
Example 2




```
CONSTRUCT { ?X foaf:name ?Y . ?X rdf:type foaf:person . }
WHERE     { ?X VCard:FN  ?Y }
```

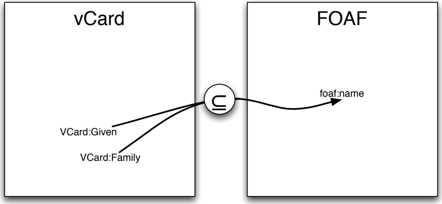
No problem either.

Example 3




```
CONSTRUCT { ?X foaf:name ???  }
WHERE     { ?X VCard:Given ?N. ?X VCard:Family ?F
          }
```

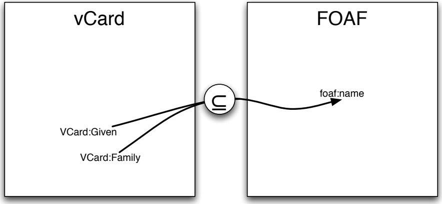
Example 3




```
CONSTRUCT { ?X foaf:name ???  }
WHERE     { ?X VCard:Given ?N. ?X VCard:Family ?F
          }
```

How to tackle? FILTERs?

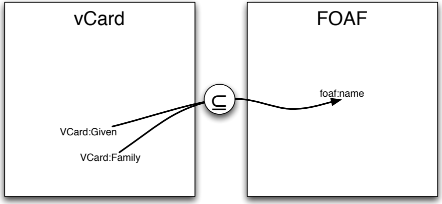
Example 3




```
CONSTRUCT { ?X foaf:name ?FN }
WHERE     { ?X VCard:Given ?N. ?X VCard:Family ?F
            FILTER( ?FN = fn:concat(?N," ",?F))}
```

Doesn't work :-| FILTERs only bind variables, can't create new bindings

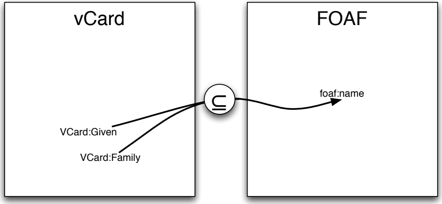
Example 3




```
CONSTRUCT { ?X foaf:name fn:concat(?N," ",?F) }
WHERE     { ?X VCard:Given ?N. ?X VCard:Family ?F
          }
```

You rather want built-in functions in the CONSTRUCT part.
This is what SPARQL++ provides.

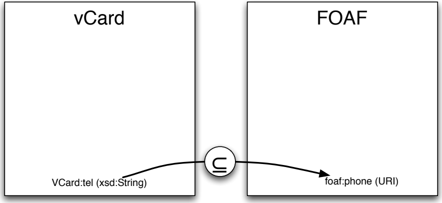
Example 3




```
CONSTRUCT { ?X foaf:name fn:concat(?N," ",?F) }
WHERE     { ?X VCard:Given ?N. ?X VCard:Family ?F
          }
```

You rather want built-in functions in the CONSTRUCT part.
This is what SPARQL++ provides.
Attention: Value generation in the CONSTRUCT part might again raise non-termination issues!

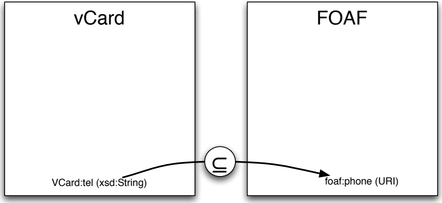
Example 4



With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
CONSTRUCT { ?X foaf:phone
   rdf:Resource(fn:concat("tel:",fn:encode-for-uri(?T)) . }
WHERE { ?X VCard:tel ?T . }
```

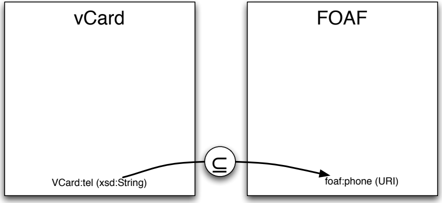
Example 4



With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
CONSTRUCT { ?X foaf:phone
   rdf:Resource(fn:concat("tel:",fn:encode-for-uri(?T)) . }
WHERE { ?X VCard:tel ?T . }
```

Example 4



With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
CONSTRUCT { ?X foaf:phone
   rdf:Resource(fn:concat("tel:",fn:encode-for-uri(?T)) . }
WHERE { ?X VCard:tel ?T . }
```

Example 5

We want more: Aggregates!

Example: Map from DOAP to RDF Open Source Software Vocabulary:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

Example 5

We want more: Aggregates!

Example: Map from DOAP to RDF Open Source Software Vocabulary:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

Example 6

Note: "Views" – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: "Import" my co-authors in my FOAF file, mapping from `myPubl.rdf` which uses the Dublin Core (DC) Vocabulary: "I know all my co-authors"

`foafWithImplicitdData.rdf`

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
WHERE { ?P rdf:type :Publ.
        ?P dc:author ?N. FILTER(?N != "Axel Polleres".)  }
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!

Example 6

Note: "Views" – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: "Import" my co-authors in my FOAF file, mapping from `myPubl.rdf` which uses the Dublin Core (DC) Vocabulary: "I know all my co-authors"

foafWithImplicitdData.rdf

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
WHERE { ?P rdf:type :Publ.
        ?P dc:author ?N. FILTER(?N != "Axel Polleres".)  }
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!

Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: “Import” my co-authors in my FOAF file, mapping from `myPubl.rdf` which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

foafWithImplicitdData.rdf

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
WHERE { ?P rdf:type :Publ.
        ?P dc:author ?N. FILTER(?N != "Axel Polleres".)  }
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!

Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

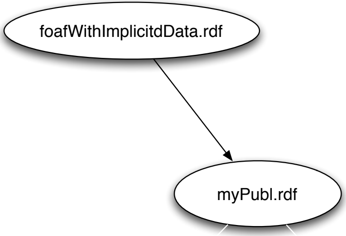
Example: “Import” my co-authors in my FOAF file, mapping from `myPubl.rdf` which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

foafWithImplicitdData.rdf

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
WHERE { ?P rdf:type :Publ.
        ?P dc:author ?N. FILTER(?N != "Axel Polleres".)  }
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

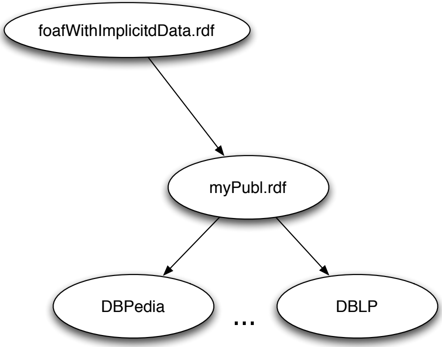
SPARQL++ allows such extended RDF Graphs!

Open Linked data with extended RDF Graphs:



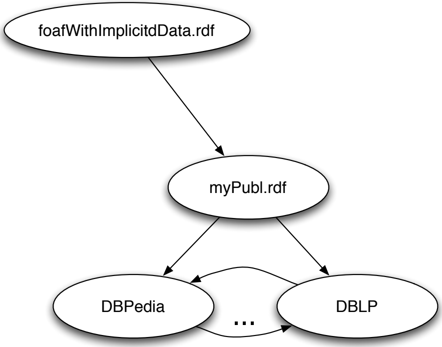
Goal: you can publish extended RDF Graphs, linked via mappings!

Open Linked data with extended RDF Graphs:



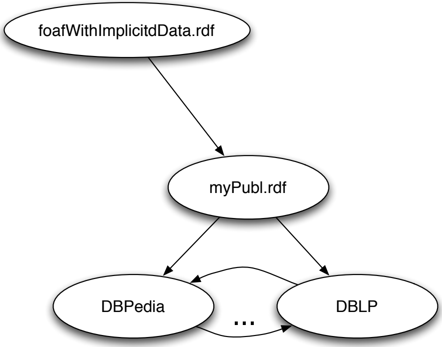
Goal: you can publish extended RDF Graphs, linked via mappings!

Open Linked data with extended RDF Graphs:



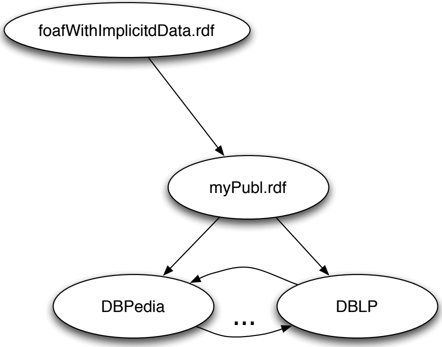
Goal: you can publish extended RDF Graphs, linked via mappings!

Open Linked data with extended RDF Graphs:



Goal: you can publish extended RDF Graphs, linked via mappings!

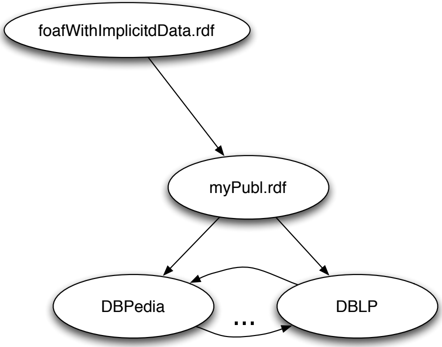
Open Linked data with extended RDF Graphs:



Goal: you can publish extended RDF Graphs, linked via mappings!

Web = HTML + Links

Open Linked data with extended RDF Graphs:



Goal: you can publish extended RDF Graphs, linked via mappings!

Semantic Web = RDF + Mappings

Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:[2]

$$h \leftarrow b_1, \ldots, b_m, \ \mathtt{not} \ \ b_{m+1}, \ldots \ \mathtt{not} \ \ b_n \qquad (1)$$

- where so-called *external atoms* of the form

$$EXT[Input](Output) \qquad (2)$$

are allowed.

- Note: External Atoms can take *predicates* as inputs → More generic than "normal" built-in predicates in logic programming!

[2]Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.

Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:[2]

$$h \leftarrow b_1, \ldots, b_m, \ \texttt{not} \ \ b_{m+1}, \ldots \ \texttt{not} \ \ b_n \tag{1}$$

- where so-called *external atoms* of the form

$$EXT[Input](Output) \tag{2}$$

are allowed.

- Note: External Atoms can take *predicates* as inputs → More generic than "normal" built-in predicates in logic programming!

[2] Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.

Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:[2]

$$h \leftarrow b_1, \ldots, b_m, \; \texttt{not} \;\; b_{m+1}, \ldots \; \texttt{not} \;\; b_n \tag{1}$$

- where so-called *external atoms* of the form

$$EXT[Input](Output) \tag{2}$$

are allowed.

- Note: External Atoms can take *predicates* as inputs → More generic than "normal" built-in predicates in logic programming!

[2]Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.

Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:[2]

$$h \leftarrow b_1, \ldots, b_m, \ \texttt{not} \ \ b_{m+1}, \ldots \ \texttt{not} \ \ b_n \qquad (1)$$

- where so-called *external atoms* of the form

$$EXT[Input](Output) \qquad (2)$$

are allowed.

- Note: External Atoms can take *predicates* as inputs → More generic than "normal" built-in predicates in logic programming!

[2] Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.

Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:[2]

$$h \leftarrow b_1, \ldots, b_m, \ \texttt{not} \ \ b_{m+1}, \ldots \ \texttt{not} \ \ b_n \qquad (1)$$

- where so-called *external atoms* of the form

$$EXT[Input](Output) \qquad (2)$$

are allowed.
- Note: External Atoms can take *predicates* as inputs → More generic than "normal" built-in predicates in logic programming!

[2] Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ...imports all RDF Triples from a given URL
- `CONCAT[`$\mathtt{Str}_1$`,...,`$\mathtt{Str}_n$`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ...returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ...returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`$\mathtt{V}_1$`,...,`$\mathtt{V}_n$`](SKTerm)` ...similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ...plus some more for handling FILTERs in SPARQL .

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ...imports all RDF Triples from a given URL
- `CONCAT[`Str_1`,...,`Str_n`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ...returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ...returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`V_1`,...,`V_n`](SKTerm)` ...similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ...plus some more for handling FILTERs in SPARQL .

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ...imports all RDF Triples from a given URL
- `CONCAT[`Str_1`,...,`Str_n`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ...returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ...returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`V_1`,...,`V_n`](SKTerm)` ...similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ...plus some more for handling FILTERs in SPARQL .

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ... imports all RDF Triples from a given URL
- `CONCAT[`Str_1`,...,`Str_n`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ... returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ... returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`V_1`,...,`V_n`](SKTerm)` ... similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ... plus some more for handling FILTERs in SPARQL .

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ...imports all RDF Triples from a given URL
- `CONCAT[`Str_1`,...,`Str_n`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ...returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ...returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`V_1`,...,`V_n`](SKTerm)` ...similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ...plus some more for handling FILTERs in SPARQL .

SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ...imports all RDF Triples from a given URL
- `CONCAT[`$\texttt{Str}_1$`,...,`$\texttt{Str}_n$`](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ...returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ...returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,`$\texttt{V}_1$`,...,`$\texttt{V}_n$`](SKTerm)` ...similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ...plus some more for handling FILTERs in SPARQL .

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Query:

```
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
WHERE { ?P a :Publ. ?P dc:author ?N.
        FILTER(?N != "Axel Polleres") }
```

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Translated HEX Program:

```
triple(S,P,O) :- &rdf["http://www.polleres.net/myPubl.rdf"](S,P,O).
```

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Translated HEX Program:

```
triple(S,P,O) :- &rdf["http://www.polleres.net/myPubl.rdf"](S,P,O).
answer(N,P) :- triple(P,"rdf:type",":Publ"),
               triple(P,"dc:author",N),
               N != "Axel Polleres".
```

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Translated HEX Program:

```
triple(S,P,O) :- &rdf["http://www.polleres.net/myPubl.rdf"](S,P,O).
answer(N,P) :- triple(P,"rdf:type",":Publ"),
               triple(P,"dc:author",N),
               N != "Axel Polleres".
triple_result(":me","foaf:knows",Blank_P) :-
   answer(N,P), &SK[ "#genid_P",N,P](Blank_P).
triple_result(Blank_P,"foaf:name",N) :-
   answer(N,P), &SK[ "#genid_P",N,P](Blank_P).
```

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Result:

```
triple_result(":me","foaf:knows","#genid_P('Francois Scharffe',:p1)")
triple_result("#genid_P('Francois Scharffe',:p1)","foaf:name","Francois Scharffe")
triple_result(":me","foaf:knows","#genid_P('Roman Schindlauer',:p1)")
triple_result("#genid_P('Roman Schindlauer',:p1)","foaf:name","Roman Schindlauer")
```

Demo Translation

Data in `myPubl.rdf`:

```
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...
```

Result:

```
triple_result(":me","foaf:knows","#genid_P('Francois Scharffe',:p1)")
triple_result("#genid_P('Francois Scharffe',:p1)","foaf:name","Francois Scharffe")
triple_result(":me","foaf:knows","#genid_P('Roman Schindlauer',:p1)")
triple_result("#genid_P('Roman Schindlauer',:p1)","foaf:name","Roman Schindlauer")
```

Can in turn be translated back to RDF Triples:

```
:me foaf:knows _:b1.
_:b1 foaf:name "Francois Scharffe".
:me foaf:knows _:b2.
_:b2 foaf:name "Roman Schindlauer".
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                         triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                         triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                          triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                   triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

$\texttt{triple\_result(P,os:latestRelease,V}_a\texttt{) :- MAX[aux}_a\texttt{,P,mask](V}_a\texttt{),}$
$\qquad\texttt{triple(P,rdf:type,doap:Project,def).}$
$\texttt{aux}_a\texttt{(P,V) :- answer}_a\texttt{(P,R,V).}$
$\texttt{answer}_a\texttt{(P,R,V) :- triple(P,doap:release R,def),}$
$\qquad\texttt{triple(R,doap:revision,V,def).}$

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                         triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                         triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                        triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                        triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                  triple(R,doap:revision,V,def).
```

aux predicate used for for projection; result of automatic translation.

Aggregates Translation:

```
CONSTRUCT { ?P os:latestRelease
   MAX(?V : ?P doap:release ?R. ?R doap:revision ?V) }
WHERE { ?P rdf:type doap:Project .  }
```

will become:

```
triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a),
                                         triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
                   triple(R,doap:revision,V,def).
```

`aux` predicate used for for projection; result of automatic translation.

Find more details on the translation in the paper.

RDFS Inference:

- RDFS Semantics can be expressed in Rules
- So, it is expressible as CONSTRUCT queries

```
CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
CONSTRUCT {?A :subClassOf ?C}
  WHERE { ?A :subClassOf ?B. ?B :subClassOf ?C. }
CONSTRUCT {?X ?B ?Y}
  WHERE { ?A :subPropertyOf ?B. ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :subClassOf ?B. ?X rdf:type ?A. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?X ?A ?Y. }
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?C :subPropertyOf ?A. ?X ?C ?Y.}
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?C :subPropertyOf ?A. ?X ?C ?Y.}
```

- Simply add these to you extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.

RDFS Inference:

- RDFS Semantics can be expressed in Rules
- So, it is expressible as CONSTRUCT queries

```
CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
CONSTRUCT {?A :subClassOf ?C}
  WHERE { ?A :subClassOf ?B. ?B :subClassOf ?C. }
CONSTRUCT {?X ?B ?Y}
  WHERE { ?A :subPropertyOf ?B. ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :subClassOf ?B. ?X rdf:type ?A. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?X ?A ?Y. }
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?C :subPropertyOf ?A. ?X ?C ?Y.}
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?C :subPropertyOf ?A. ?X ?C ?Y.}
```

- Simply add these to you extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.

RDFS Inference:

- RDFS Semantics can be expressed in Rules
- So, it is expressible as CONSTRUCT queries

```
CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
CONSTRUCT {?A :subClassOf ?C}
  WHERE { ?A :subClassOf ?B. ?B :subClassOf ?C. }
CONSTRUCT {?X ?B ?Y}
  WHERE { ?A :subPropertyOf ?B. ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :subClassOf ?B. ?X rdf:type ?A. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?X ?A ?Y. }
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?X ?A ?Y. }
CONSTRUCT {?X rdf:type ?B}
  WHERE { ?A :domain ?B. ?C :subPropertyOf ?A. ?X ?C ?Y.}
CONSTRUCT {?Y rdf:type ?B}
  WHERE { ?A :range ?B.  ?C :subPropertyOf ?A. ?X ?C ?Y.}
```

- Simply add these to you extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.

Outline

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don't provide the right "ingredients" to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of "Extended Graphs", i.e. Mappings+RDF Data in one file, similar to "Networked Graphs" [Schenk and Staab, 2007][3]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A "safety condition" for recursive mappings with bnodes and value-generating CONSTRUCTs.

[3] diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-)

Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic **Web**...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g. http://pipes.deri.org, http://www.sindice.com, dlvhex-server

Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-)

References I

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/.

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.

Polleres, A., Scharffe, F., and Schindlauer, R. (2007).
SPARQL++ for mapping between RDF vocabularies.
In *OTM 2007, Part I : Proceedings of the 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 878–896, Vilamoura, Algarve, Portugal. Springer.

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.