dlvhex-sparql: A SPARQL-compliant Query Engine based on dlvhex *

Axel Polleres[1] and Roman Schindlauer[2]

[1] DERI Galway, National University of Ireland, Galway
axel@polleres.net
[2] Univ. della Calabria, Rende, Italy and Vienna Univ. of Technology, Austria
roman@kr.tuwien.ac.at

Abstract. This paper describes the dlvhex SPARQL plugin, a query processor for the upcoming Semantic Web query language standard by W3C. We report on the implementation of this languages using dlvhex, a flexible plugin system on top of the DLV solver. This work advances our earlier translation based on the semantics by Perez et al. towards an engine which is fully compliant to the official SPARQL specification. As it turns out, the differences between these two definitions of SPARQL, which might seem moderate at first glance, need some extra machinery. We also briefly report the status of implementation, and extensions currently being implemented, such as handling of aggregates, nested CONSTRUCT queries in the spirit of networked RDF graphs, or partially support of RDFS entailment. For such extensions a tight integration of SPARQL query processing and Answer-Set Programming, the underlying logic programming formalism of our engine, turns out to be particularly useful, as the resulting programs can actually involve unstratified negation.

1 Introduction

SPARQL, the upcoming Semantic Web query language, is short before being standardized by the W3C, and has just reached Candidate Recommendation Status [8]. As opposed to earlier versions of this specification, the formal underpinnings of the language have been seriously improved, influenced by results from academia such as Perez et al.'s work [6]. In [7] we presented a translation from SPARQL to Datalog following Perez et al. and showed how we can cover even corner-cases such as non-well-designed query patterns, where UNION and OPTIONAL patterns turned out to be particularly tricky. In the present work we aim at bridging the gap between the formal translation from [7] towards an actual implementation of the official W3C candidate recommendation. Compared with the semantics presented in [6, 7], the recent specification shows some differences which require additional machinery, such as the treatment of filters in optional graph patterns, multiset semantics, and the handling of blank nodes in CONSTRUCT queries which have an impact for practical implementations. This paper is to

* This work has been supported by the European FP6 projects inContext (IST-034718) and REWERSE (IST 506779), by the Austrian Science Fund (FWF) project P17212-N04, by the AECI Programa de Cooperación Interuniversitaria e Investigación Científica entre España y los pases de Iberoamérica (PCI), by the Consejería de Educación de la Comunidad de Madrid and Universidad Rey Juan Carlos under the project URJC-CM-2006-CET-0300, as well as by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131).

be conceived as a system description: Rather by use of practical examples than repeating formal details from our earlier works, we will show how our earlier translation can be lifted to a more spec-compliant one. Moreover, we report on implementation details of our prototypical engine dlvhex-sparql.

We will review the basics of dlvhex and main ideas of our translation by means of simple examples in Section 2. In Subsection 2.3 we will discuss the main differences between our original semantics from [7] and the current SPARQL specification [8] along with patches for our translation. Next, we will present some details about our prototype implementation in Section 3. Finally, in Section 4 we will motivate further why a tight integration of SPARQL query processing and answer-set programming, the underlying logic programming of our engine, turns out to be particularly useful and will give an outlook to future work.

2 From SPARQL to dlvhex

As shown in [7] the semantics of SPARQL SELECT queries can, to a large extent, be translated to Datalog programs with minimal support of built-in predicates. Hence, any logic programming engine which supports Datalog (i.e., function-free) with negation as failure, as well as built-in functions to import triples from given RDF graphs, could in principle serve as a SPARQL engine. We will focus here particularly on our implementation of this translation using the dlvhex engine, a flexible and extensible plugin framework on top of the DLV system to support a wide range of external predicates.

2.1 dlvhex Basics

dlvhex [3] is a reasoner for so-called HEX-programs [11], a relatively new logic programming language, which provides an interface to external sources of knowledge. The definition of this interface is very general, allowing for the implementation of a wide range of specialized tasks, such as the import of RDF data, basic string manipulation routines, or even aggregate functions.

The crucial feature of HEX-programs are *external atoms*, which are of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m),$$

where $Y_1, \ldots, Y_n$ is a list of predicates and terms and $X_1, \ldots, X_m$ is a list of terms (called *input list* and *output list*, respectively), and g and *output arities* n and m fixed for g. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates and terms. Note that this means that external predicates, unlike usual definitions of built-ins in logic programming, can not only take constant parameters but also (extensions of) predicates as input.

A *rule* is of the form

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

where h and b_i ($1 \leq i \leq n$) are atoms, b_k ($1 \leq k \leq m$) are either atoms or external atoms, and 'not' is the symbol for negation as failure.

The semantics of dlvhex generalizes the well-known answer-set semantics [4] by extending it to external atoms. One distinguished feature of the answer-set semantics is

[3] Available on http://www.kr.tuwien.ac.at/research/dlvhex/.

its ability to generate multiple minimal models for a single problem specification. Its treatment of negation as failure qualifies it as an intuitive way to deal with unstratified negation in logic programming. Answer-set programming is particularly suitable for combinatorial search problems and their applications.

In our implementation we translate SPARQL queries to HEX-programs with a set of dedicated external atoms, only two of which we mention here explicitly. Further external atoms and built-in functions are necessary to deal with complex FILTER expressions as defined in [8, Sec. 11.3], where we refer to [7, 9] for further details.

RDF Import The access on RDF knowledge is realized through the `&rdf` predicate. It is of the form $\&rdf[i](s, p, o)$, where both the input term i as well as the output terms s, p, o are constants The external atom $\&rdf[i](s, p, o)$ is true if (s, p, o) is an RDF triple *entailed* by the RDF graph which is accessibly at IRI i. Here, we consider simple RDF entailment [5] only.

Skolemizing Blank Nodes In order to properly deal with blank nodes in CONSTRUCTs (see Subsection 2.3), we need to be able to generate fresh blank node identifiers. The idea here is similar to Skolemization. The external predicate $\&\mathtt{sk}[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

As widely known for programs without external predicates, safety [12] guarantees that the number of entailed ground atoms is finite. Though, by external atoms in rule bodies, new, possibly infinitely many, ground atoms could be generated, even if all atoms themselves are safe. In order to avoid this, the notion of *strong safety* [11] for HEX-programs, constrains the use of external atoms in cyclic rules and guarantees finiteness of models as well as finite computability of external atoms.

2.2 From SPARQL to dlvhex by Example

In this section, we exemplify our translation by means of some illustrating sample SPARQL queries. We assume basic familiarity of the reader with RDF and SPARQL, and will only briefly intoduce some basics here: We define a SPARQL *query* as a tuple $Q = (R, P, DS)$ where R is a result form, P a graph pattern, and DS a dataset.[4] For a SELECT query, a *result form* R is simply a set of variables, whereas for a CONSTRUCT query, the result form R is a set of triple patterns.

We assume the pairwise disjoint, infinite sets I, B, L and Var, which denote IRIs, blank node identifiers, RDF literals,[5] and variables respectively.
Graph patterns are recursively defined as follows:

- $s\ p\ o.$ is a graph pattern where $s, o \in I \cup B \cup L \cup Var$ and $p \in I \cup Var$.
- A set of graph patterns is a graph pattern.
- Let P, P_1, P_2 be graph patterns, R a filter expression, and $i \in I \cup Var$, then P_1 OPTIONAL P_2,P_1 UNION P_2,GRAPH i P, and P FILTER R are graph patterns.

For any pattern P, we denote by $vars(P)$ the set of all variables occurring in P and by $\overline{vars}(P)$ the tuple obtained by the lexicographic ordering of all variables in P. As *atomic filter expression*, we allow here the unary predicates BOUND (possibly with

[4] We will ignore solution modifiers for the purpose of this paper, since they can be added by post-processing results of our translation.

[5] For sake of brevity, we only cover plain (i.e., untyped, not language tagged) literals here.

variables as arguments), isBLANK, isIRI, isLITERAL, and binary comparison predicates '$=$', '$<$', '$>$' with arbitrary safe built-in terms as arguments. *Complex filter expressions* can be built using the connectives '$\neg$', '$\wedge$', and '$\vee$'.

The *dataset* $DS = (G, \{(g_1, G_1), \ldots, (g_k, G_k)\})$ of a SPARQL query is defined by a default graph G plus a set of named graphs, i.e., pairs of IRIs and corresponding graphs. Without loss of generality (there are other ways to define the dataset such as in a SPARQL protocol query), we assume G given as the merge of the graphs denoted by the IRIs explicitly given in a set of FROM clauses and the named graphs $g_1, \ldots, g_k$ are specified in the form of FROM NAMED clauses.

For the following example queries, we assume the datasets consisting of two RDF graphs with the URIs `http://alice.org` and `http://ex.org/bob` which contain some information about Alice and Bob encoded in the commonly used FOAF[6] vocabulary. For instance, the following SELECT query $Q = (R, P, DS)$ selects all persons who know somebody, and the names of these persons.[7]

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

Here, P is a simple set of triple patterns, also called *basic graph pattern* in SPARQL. R is the set of variables $\{?X, ?Y\}$, $DS = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, i.e., the default graph being the merge of the two graphs and an empty set of named graphs, since no FROM NAMED clause is given. Using the RDF-plugin of dlvhex such queries can be translated to a simple HEX-program:[8]

```
(1)  tripleQ(S,P,O,def)   :-  &rdf["http://ex.org/bob"](S,P,O).
(2)  tripleQ(S,P,O,def)   :-  &rdf["http://alice.org"](S,P,O).
(3)  answerP(X,Y,BLANK_x,DS)  :-  triple(X,rdf:type,foaf:Person,DS),
                                  triple(X,foaf:name,Y,DS),
                                  triple(X,foaf:knows,BLANK_x,DS).
(4)  answerQ(X,Y)   :-  answerP(X,Y,BLANK_x,def).
```

Here, rules (1)+(2) "collect" the dataset by merging the two source graphs in the predicate $\texttt{triple}_Q$, where the constant `def` in last parameter denotes the triples of the default graph. Disambiguation of possible overlapping blank node ids in the source graphs is taken care of by the RDF plugin, i.e., during import the `&rdf` predicate gives a fresh id to any blank node. As we can see in rule (3), basic graph patterns basically boil down to simple conjunctive queries over the predicate $\texttt{triple}_Q$ of which the results are collected in the predicate $\texttt{answer}_P(\overline{vars}(P), \texttt{DS})$. The variable `DS` denotes the part of the dataset the pattern refers to (see the next example for more details). Blank nodes in P are simply treated as special variables, which is a quite standard procedure (see e.g., [3].[9] The projection to the variables in R and restriction to results from the default graph takes place in rule (4), which finally collects all solution tuples for the query in the dedicated predicate $\texttt{answer}_Q$.

[6] `http://xmlns.com/foaf/spec/`

[7] As usual in SPARQL or Turtle [2], the predicate '`rdf:type`' is abbreviated with '`a`'.

[8] dlvhex uses the common PROLOG style notation where unquoted uppercase terms denote variables and all other terms denote constants. We simplify here from the notation used in the actual implementation, where some encoding is necessary in order to distinguish RDF literals, IRIs, blank nodes, etc.

[9] For the treatment of blank nodes in R, i.e., in CONSTRUCTs, we refer to Section 2.3 below.

More complex, possibly nested patterns are handled by introducing, for each subpattern of P, new auxiliary predicates $\texttt{answer1}_P$, $\texttt{answer2}_P$, $\texttt{answer3}_P$, etc. We exemplify this by the following GRAPH query which selects creators of graphs and the persons they know.

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

This query is translated to a HEX-program as follows:

```
(1) tripleQ(S,P,O,def)  :-  &rdf["http://alice.org"](S,P,O).
(2) tripleQ(S,P,O,"http://alice.org")  :-  &rdf["http://alice.org"](S,P,O).
(3) tripleQ(S,P,O,"http://ex.org/bob")  :-  &rdf["http://ex.org/bob"](S,P,O).
(4) answer1P(X,Y,DS)  :-  tripleQ(G,foaf:maker,X,DS), answer2(X,Y,G), G != def.
(5) answer2P(X,Y,DS)  :-  tripleQ(X,foaf:knows,Y,DS),
                          tripleQ(X,rdf:type,foaf:Person,DS).
(6) answerQ(X,Y)  :-  answer1P(X,Y,def).
```

Here, again the first rules (1)-(3) import the dataset, now also involving named graphs. The GRAPH subpattern is computed by predicate $\texttt{answer2}_P$, and we see that the last parameter in the triple predicate carries over bindings to particular named graphs or via the constant `def` to the default graph. Note that the inequality atom `G != def` in rule (4) serves to restrict answers for the GRAPH subpattern to only refer to named graphs, according to SPARQL's semantics.

Next, let us turn to a query that involves a UNION pattern, asking for persons and their names *or* nicknames.

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

Alternatives can be modeled by splitting off the branches in a UNION pattern into several rules with the same `answer` head predicate:

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(X,Y,Z,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS), answer2P(X,Y,Z,DS).
(3) answer2P(X,Y,null,DS)  :-  tripleQ(X,foaf:name,Y,DS).
(4) answer2P(X,null,Z,DS)  :-  tripleQ(X,foaf:nick,Z,DS).
(6) answerQ(X,Y,Z)  :-  answer1P(X,Y,Z,def).
```

Since we bind names and nicknames to different variables Y and Z here, the answers for the non-occurring variable will be unbound in the respective branch of the UNION. We emulate such unboundedness in our translation by null values [7] in the rules (3)+(4).

Let us turn to OPTIONAL patterns by the following example query which selects all persons and optionally their names:

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

OPTIONALs can be emulated again by null values and using negation as failure.

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(N,X,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS), answer2P(N,X,DS).
(3) answer1P(null,X,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS),
                             not answer2'P(X,DS).
(4) answer2P(N,X,DS)  :-  tripleQ(X,foaf:name,N,DS).
(5) answer2'P(X,DS)  :-  answer2P(N,X,DS).
(6) answerQ(N,X)  :-  answer1P(N,X,def).
```

In rules (3)+(5) we cover the case where the optional part has no solutions for X by a rule with head predicate $\texttt{answer2'}_P$ which projects away all variables only occurring in the optional part ($\texttt{answer2}_P$) and which we negate in rule (3).

As for the treatment of FILTER expressions, we made the restricting assumption in [7] that each variable appearing in a FILTER expression needs to be bound in a triple pattern in the same scope as the FILTER expression, since otherwise our recursive translation given in [7] would construct possible unsafe rules. Take for instance the pattern $P = \{$ `?X foaf:mbox ?M . FILTER(?Age > 30)` $\}$. In our translation without further modification, such a pattern this would yield a rule like:

$$\texttt{answer}_P\texttt{(Age,M,X,DS)} \texttt{ :- } \texttt{triple}_Q\texttt{(X,foaf:mbox,M,DS), Age > 30.}$$

where `Age` is an unsafe variable occurring in a built-in atom. however, it turns out that the safety restriction for variables in FILTERs is unnecessary, since we could remedy the situation by replacing all unsafe variables in a FILTER simply by the constant null again, which yields for our example pattern P:

$$\texttt{answer}_P\texttt{(null,M,X,DS)} \texttt{ :- } \texttt{triple}_Q\texttt{(X,foaf:mbox,M,DS), null > 30.}$$

As expected, this rule can never fire, since the built-in atom `null > 30` is always false.

Finally, let us turn our attention to CONSTRUCT queries. We suggested in [7] that we can allow CONSTRUCT queries of the form $Q = (R, P, DS)$ where R consists of bNode-free triple patterns. We can model these by adding a rule

$$\texttt{triple}_Q\texttt{(}s,p,o\texttt{,res)} \texttt{ :- } \texttt{answer}_Q\texttt{(}\overline{vars}(P)\texttt{)}.$$

for each triple pattern $s\ p\ o.$ in R[10] to the translated program. The result graph `res` is then naturally represented in the answer set of the extended program, namely by those tuples in the extension of the predicate $\texttt{triple}_Q$ having `res` as the last parameter and representing valid RDF triples.

Apart from some extra-machinery, which is needed in the case of non-well-designed graph patterns [6], the examples in this section should cover the basic ideas behind our translation which had been described in [7], and we refer the interested reader to this work for further details. The present paper is rather focused on implementation specific aspects concerning the latest official specification of SPARQL and some difficulties which arise from particular decisions taken by the W3C Data Access working group. We will cover these issues in the next subsection.

2.3 Full SPARQL Compliance

In order to arrive at a SPARQL compliant translation, we face the following difficulties:

1. How to deal with solution modifiers.
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.

[10] Analogously to the FILTER example, we can replace variables unbound in P but occurring in R by null again in order to ensure safety.

As for 1, we do not yet treat solution modifiers such as ORDER BY and OFFSET in our current prototype, but these can be easily added by post-processing the results obtained from our translation fed into dlvhex. Issue 2 is somewhat harder to solve. Note that our current translation, as well as the SPARQL semantics defined by Perez et al. [6] creates sets of solutions, i.e., each query is treated as if it was a DISTINCT query. Take for instance a variation of our UNION example from above:

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

and assume the source graph

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

The naive translation of the above query to a HEX-program is as follows:

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(N,X,DS)  :-  tripleQ(X,foaf:name,N,DS).
(3) answer1P(N,X,DS)  :-  tripleQ(X,foaf:nick,N,DS).
(4) answerQ(N)  :-  answer1P(N,X,def).
```

This program (in a bottom-up evaluation such as the one underlying the dlvhex system) would result in two answers $\texttt{answer}_Q\texttt{("Bob")}$ and $\texttt{answer}_Q\texttt{("Bobby")}$. According to the official SPARQL semantics, however, the above query has four solutions binding variable `?N` three times to `"Bob"` and twice to `"Bobby"`. If we observe where the duplicates get "lost" in our translation, we can see that only (i) the final projection in predicate $\texttt{answer}_Q$ and (ii) duplicates due to UNION patterns cause us to lose duplicates. We can remedy this easily by (i) always carrying over all the variables in all subpatterns to the $\texttt{answer}_Q$ predicate and only projecting out the non-selected variables during postprocessing, and (ii) adding an extra variable for each UNION pattern which models possible branches a solution stems from. The such modified version of our translated program looks as follows:

```
(1)  tripleQ(S,P,O,def)  :-  ...
(2') answer1P(N,X,1,DS)  :-  tripleQ(X,foaf:name,N,DS).
(3') answer1P(N,X,2,DS)  :-  tripleQ(X,foaf:nick,N,DS).
(4') answerQ(N,X,Union1)  :-  answer1P(X,N,Union1,def).
```

Here, the constants `1` and `2` mark the branches of the union in rules (2')+(3'), and are carried over to the end result in rule (4') by the extra variable `Union1`. Indeed, this modified program has four answers $\texttt{answer}_Q\texttt{("Bob",:bob,1)}$, $\texttt{answer}_Q\texttt{("Bobby",:bob,2)}$, $\texttt{answer}_Q\texttt{("Bobby",\_:a,2)}$, and $\texttt{answer}_Q\texttt{("Bob",\_:a,2)}$.

Regarding issue 3, let us consider a query involving the above mentioned FILTER condition:

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

Here, we want to select names of persons and only output email addresses (`foaf:mbox`) of those ones older than 30. The possibility of FILTERs within OPTIONALs to refer to variables bound outside the enclosing OPTIONAL pattern is an interesting feature of SPARQL for such queries, however, our original translation would treat filters strictly local to their pattern:

```
(1)  triple_Q(S,P,O,def)  :-  ...
(2)  answer1_P(Age,N,M,X,DS)  :-  triple_Q(X,foaf:name,N,DS), triple_Q(X,:age,Age,DS),
                                  answer2_P(Age,M,X,DS).
(3)  answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                     triple_Q(X,:age,Age,DS),
                                     not answer2'_P(Age,X,DS).
(4)  answer2_P(null,M,X,DS)  :-  triple_Q(X,foaf:mbox,M,DS), null > 30.
(5)  answer2'_P(Age,X,DS)  :-  answer2_P(Age,M,X,DS).
(6)  answer_Q(N,M)  :-  answer1_P(Age,N,M,X,def).
```

Since the use of variable `Age` in rule (4) would be unsafe, our original translation replaces it by null, thus not returning any email addresses (i.e., bindings for `N`) for the overall query. The solution is now to modify the translation in order to draw FILTERs in the scope of OPTIONALs upwards in the pattern tree, yielding a modified translation:

```
(1)  triple_Q(S,P,O,def)  :-  ...
(2') answer1_P(Age,N,M,X,DS)  :-  triple_Q(X,foaf:name,N,DS), triple_Q(X,:age,Age,DS),
                                  answer2_P(M,X,DS), Age > 30.
(3a') answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                      triple_Q(X,:age,Age,DS),
                                      answer2_P(M,X,DS), not Age > 30.
(3b') answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                      triple_Q(X,:age,Age,DS), not answer2'_P(X,DS).
(4') answer2_P(M,X,DS)  :-  triple_Q(X,foaf:mbox,M,DS).
(5') answer2'_P(X,DS)  :-  answer2_P(M,X,DS).
(6') answer_Q(N,M)  :-  answer1_P(Age,N,M,X,def).
```

Rules (2')-(3b') now exactly reflect the case distinction for OPTIONALs by the definition of the LeftJoin operator in [8, Section 12.4]. As an interesting side-note, we remark that the non-local behavior of filter expressions only applies to FILTERs on the top level of OPTIONALs: The reader might easily convince herself by the definitions in the current SPARQL specification that a slightly modified query

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

is not semantically equivalent to the original query although the triple `?X foaf:name ?N` inside the OPTIONAL seems to be redundant at first glance. In fact, the difference here is that FILTERs which are nested within a group graph pattern will be evaluated local to this pattern, not taking bindings from outside the OPTIONAL into account.

Finally, let us turn to issue 4, namely the translation of CONSTRUCT queries involving blank nodes in the result form. We consider an example query which constructs `foaf:maker` relations for people authoring a document, expressed by the Dublin Core property `dc:creator`. We assume that in the source graph all values for `dc:creator` are literals denoting the authors' names. Thus, we want to create bNodes for each author, since the `foaf:maker` of a document should be a `foaf:Agent`:

```
CONSTRUCT { _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b. } FROM ...
    WHERE { ?Doc dc:creator ?N. }
```

The idea to implement the SPARQL semantics properly here is to use the external predicate `&sk` mentioned in Subsection 2.1 to generate new blank node identifiers for each solution binding for $\overline{var}(P)$ similar in spirit to Skolemization . We simply use the original bNode identifier `b` in R as "Skolem function":

```
(1) triple_Q(S,P,O,def)  :-  ...
(2) answer1_P(Doc,N,DS)  :-  triple_Q(Doc,dc:creator,N,DS).
(3) triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1_P(Doc,N,def),
                                                     &sk[b,Doc,N](BLANK_b).
(4) triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1_P(Doc,N,def),
                                             &sk[b,Doc,N](BLANK_b).
(5) triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1_P(Doc,N,def),
                                                &sk[b,Doc,N](BLANK_b).
```

Note that, since we use different predicates $\texttt{triple}_{Res}$ and $\texttt{triple}_{Q}$for the result triples and dataset triples here, the resulting program stays in principle non-recursive and thus strong safety as discussed in Subsection 2.1 is guaranteed, despite the generation of new values by means of the external predicate `&sk`.

3 Prototype Implementation

We implemented a prototype of a SPARQL engine based on the dlvhex solver, called dlvhex-sparql. The external atoms in HEX-programs are provided by so-called *plugins*, which are dynamically loaded at run-time by the evaluation framework of dlvhex. A plugin may also supply a rewriting module, which is executed prior to the model generation algorithm and allows for a conversion of the input data into a valid HEX-program. The prototype exploits the rewriting mechanism of the dlvhex framework, taking care of the translation of a SPARQL query into the appropriate HEX-program, as laid out in Subsection 2.2. The system implements external atoms used in the translation, namely (i) the `&rdf`-atom for data import aggregate atoms, and (ii) a string manipulation atom implementing the `&sk`-atom for blank node handling. The default syntax of a dlvhex results corresponds to the usual answer format of logic programming engines, i.e., sets of facts, from which we generate an XML representation that can subsequently be transformed easily to a valid RDF syntax by an XSLT to export solution graphs.

Note that the support of complex FILTER expressions is only rudimentary at the moment and subject to ongoing work. As mentioned before, we will need a dedicated set of additional external atoms in order to support the full extent of FILTER expressions as described in [8, Sec. 11.3].

We also implemented a rudimentary Web service interface making our engine accessible as a general purpose SPARQL endpoint. This was realized by XSL transforming the XML output of dlvhex into the result format prescribed by the SPARQL protocol[11] and is accessible via a SOAP interface at `http://apolleres.escet.urjc.es:8080/axis/services/SparqlEvaluator?wsdl`.

4 Extensions and Next Steps

While the current implementation efforts around dlvhex-sparql described here were focused on conceptually proving the feasibility of a fully SPARQL compliant query engine on top of dlvhex, our intentions behind go well beyond this sheer exercise. We are currently working on extensions such as allowing aggregate and built-in functions in the result form of queries, which allows computations of new values. Such an extension is crucial for instance for mapping between different, overlapping RDF vocabularies [1]. In this context, we plan to support the use of CONSTRUCT queries as part of the dataset which allows to express such mappings[12] or interlinked, implicit RDF metadata[13]. The embedding of such extensions into our translation comes mostly without

[11] `http://www.w3.org/TR/2006/CR-rdf-sparql-protocol-20060406/`

[12] `www.rdfweb.org/topic/ExpertFinder_2fmappings`

[13] `www.w3.org/2005/rules/wg/wiki/UCR/Publishing_Rules_for_Interlinked_Metadata`

additional costs, since the respective query translations for both the actual query as well as mapping rules and views in the form of CONSTRUCTs can be translated into a single dlvhex program and evaluated at once. Here is where the power of answer-set programming comes into play, since such combined programs may involve unstratified recursion which can be dealt with flexibly under brave or cautions reasoning, respectively. We should mention here related approaches such as [10], which alternatively suggest the use of the well-founded semantics for such scenarios, but with a similar intention to create networks of RDF graphs (possibly recursively) linked by CONSTRUCT queries. Moreover, we did not yet conduct extensive performance evaluations, but we would not expect to be necessarily competitive with special-purpose SPARQL engines. However, the power of our approach lies in its natural combination of RDF with the rules world, which for instance allows us to plug-in on the fly Datalog rulesets which emulate RDF(S) entailment (see for instance[3]).

References

1. B. Aleman-Meza, U. Bojars, H. Boley, J. G. Breslin, M. Mochol, L. J. Nixon, A. Polleres, and A. V. Zhdanova. Combining RDF vocabularies for expert finding. In *Proceedings of the 4th European Semantic Web Conference (ESWC2007)*, number 4519 in Lecture Notes in Computer Science, pages 235–250, Innsbruck, Austria, June 2007. Springer.
2. D. Beckett. Turtle - Terse RDF Triple Language, Apr. 2006. Available at `http://www.dajobe.org/2004/01/turtle/`.
3. J. de Bruijn and S. Heymans. RDF and logic: Reasoning and extension. In *Proceedings of the 6th International Workshop on Web Semantics (WebS 2007), in conjunction with the 18th International Conference on Database and Expert Systems Applications (DEXA 2007)*, Regensburg, Germany, September 3–7 2007. IEEE Computer Society Press.
4. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
5. P. Hayes. RDF semantics. Technical report, W3C, February 2004. W3C Recommendation.
6. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference (ISWC 2006)*, pages 30–43, 2006.
7. A. Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th World Wide Web Conference (WWW2007)*, Banff, Canada, May 2007.
8. E. Prud'hommeaux and A. Seaborne (eds.). SPARQL Query Language for RDF, June 2007. W3C Candidate Recommendation, available at `http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/`.
9. S. Schenk. A SPARQL Semantics Based on Datalog. In *KI2007, Osnabrck, Germany*, 2007.
10. S. Schenk and S. Staab. Networked RDF Graphs. Tech. rep., Univ. Koblenz, 2007. `http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf`.
11. R. Schindlauer. *Answer-Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Dec. 2006.
12. J. Ullman. *Principles of Database & Knowledge Base Systems*. Comp. Science Press, 1989.