# SPARQL++ for Mapping between RDF Vocabularies[⋆]

Axel Polleres[1], François Scharffe[2], and Roman Schindlauer[3,4]

[1] DERI Galway, National University of Ireland, Galway
axel@polleres.net
[2] Leopold-Franzens Universität Innsbruck, Austria
francois.scharffe@uibk.ac.at
[3] Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
[4] Institut für Informationssysteme, Technische Universität Wien
roman@kr.tuwien.ac.at

**Abstract.** Lightweight ontologies in the form of RDF vocabularies such as SIOC, FOAF, vCard, etc. are increasingly being used and exported by "serious" applications recently. Such vocabularies, together with query languages like SPARQL also allow to syndicate resulting RDF data from arbitrary Web sources and open the path to finally bringing the Semantic Web to operation mode. Considering, however, that many of the promoted lightweight ontologies overlap, the lack of suitable standards to describe these overlaps in a declarative fashion becomes evident. In this paper we argue that one does not necessarily need to delve into the huge body of research on ontology mapping for a solution, but SPARQL itself might — with extensions such as external functions and aggregates — serve as a basis for declaratively describing ontology mappings. We provide the semantic foundations and a path towards implementation for such a mapping language by means of a translation to Datalog with external predicates.

## 1 Introduction

As RDF vocabularies like SIOC,[5] FOAF,[6] vCard,[7] etc. are increasingly being used and exported by "serious" applications we are getting closer to bringing the Semantic Web to operation mode. The standardization of languages like RDF, RDF Schema and OWL has set the path for such vocabularies to emerge, and the recent advent of an operable query language, SPARQL, gave a final kick for wider adoption. These ingredients allow not only to publish, but also to syndicate and reuse metadata from arbitrary distibuted Web resources in flexible, novel ways.

When we take a closer look at emerging vocabularies we realize that many of them overlap, but despite the long record of research on ontology mapping and alignment, a standard language for defining mapping rules between RDF vocabularies is still

⋆ This research has been partially supported by the European Commission under the FP6 projects inContext (IST-034718), REWERSE (IST 506779), and Knowledge Web (FP6-507482), by the Austrian Science Fund (FWF) under project P17212-N04, as well as by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131).

[5] http://sioc-project.org/

[6] http://xmlns.com/foaf/0.1/

[7] http://www.w3.org/TR/vcard-rdf

missing. As it turns out, the RDF query language SPARQL [24] itself is a promising candidate for filling this gap: Its CONSTRUCT queries may themselves be viewed as rules over RDF. The use of SPARQL as a rules language has several advantages: (i) the community is already familiar with SPARQL's syntax as a query language, (ii) SPARQL supports already a basic set of built-in predicates to filter results and (iii) SPARQL gives a very powerful tool, including even non-monotonic constructs such as OPTIONAL queries.

When proposing the use of SPARQL's CONSTRUCT statement as a rules language to define mappings, we should first have a look on existing proposals for syntaxes for rules languages on top of RDF(S) and OWL. For instance, we can observe that SPARQL may be viewed as syntactic extension of SWRL [16]: A SWRL rule is of the form $ant \Rightarrow cons$, where both antecedent and consequent are conjunctions of atoms $a_1 \wedge \ldots \wedge a_n$. When reading these conjunctions as basic graph patterns in SPARQL we might thus equally express such a rule by a CONSTRUCT statement:

$$\text{CONSTRUCT } \{ \ cons \ \} \text{ WHERE } \{ \ ant \ \}$$

In a sense, such SPARQL "rules" are more general than SWRL, since they may be evaluated on top of arbitrary RDF data and — unlike SRWL — not only on top of valid OWL DL. Other rules language proposals, like WRL [7] or TRIPLE [8] which are based on F-Logic [18] Programming may likewise be viewed to be layerable on top of RDF, by applying recent results of De Bruijn et al. [5, 6]. By the fact that (i) expressive features such as negation as failure which are present in some of these languages are also available in SPARQL [8] and (ii) F-Logic molecules in rule heads may be serialized in RDF again, we conjecture that rules in these languages can similarly be expressed as syntactic variants of SPARQL CONSTRUCT statements.[9]

On the downside, it is well-known that even a simple rules language such as SWRL already lead to termination/undecidability problems when mixed with ontology vocabulary in OWL without care. Moreover, it is not possible to express even simple mappings between common vocabularies such as FOAF[6] and VCard[7] in SPARQL only. To remedy this situation, we propose the following approach to enable complex mappings over ontologies: First, we keep the expressivity of the underlying ontology language low, restricting ourselves to RDFS, or, more strictly speaking to, $\rho$df$^-$ ontologies (a variant of $\rho$df [20] defined in Subsection 4.5); second, we extend SPARQL's CONSTRUCT by features which are almost essential to express various mappings, namely: a set of useful built-in functions (such as string-concatenation and arithmetic functions on numeric literal values) and aggregate functions (min, max, avg). Third, we show that evaluating SPARQL queries on top of $\rho$df$^-$ ontologies plus mapping rules is decidable by translating the problem to query answering over HEX-programs, i.e., logic programs with external built-ins using the answer-set semantics, which gives rise to implementations on top of existing rules engines such as dlvhex. A prototype of a SPARQL engine for evaluating queries over combined datasets consisting of $\rho$df$^-$ and SPARQL mappings has been implemented and is avaiblable for testing online.[10]

[8] see [24, Section 11.4.1]

[9] with the exception of predicates with arbitrary arities

[10] `http://kr.tuwien.ac.at/research/dlvhex/`

The remainder of this paper is structured as follows. We start with some motivating examples of mappings which can and can't be expressed with SPARQL CONSTRUCT queries in Section 2 and suggest syntactic extensions of SPARQL, which we call SPARQL++, in order to deal with the mappings that go beyond. In Section 3 we introduce HEX-programs, whereafter in Section 4 we show how SPARQL++ CONSTRUCT queries can be translated to HEX-programs, and thereby bridge the gap to implementations of SPARQL++. Next, we show how additional ontological inferences by $\rho$df$^-$ ontologies can be itself viewed as a set of SPARQL++ CONSTRUCT "mappings" to HEX-programs and thus embedded in our overall framework, evaluating mappings and ontological inferences at the same level, while retaining decidability. After a brief discussion of our current prototype and a discussion of related approaches, we conclude in Section 6 with an outlook to future work.

## 2 Motivating Examples – Introducing SPARQL

Most of the proposals in the literature for defining mappings between ontologies use subsumption axioms (by relating defining classes or (sub)properties) or bridge rules [3]. Such approaches do not go much beyond the expressivity of the underlying ontology language (mostly RDFS or OWL). Nonetheless, it turns out that these languages are insufficient for expressing mappings between even simple ontologies or when trying to map actual sets of data from one RDF vocabulary to another one. In Subsection 10.2.1 of the latest SPARQL specification [24] an example for such a mapping from FOAF to VCard is explicitly given, translating the VCard properties into the respective FOAF properties most of which could equally be expressed by simple rdfs:subPropertyOf statements. However, if we think the example a bit further, we quickly reach the limits of what is expressible by subclass- or subproperty statements.

*Example 1.* A simple and straightforward example for a mapping from `VCard:FN` to `foaf:name` is given by the following SPARQL query:

```
CONSTRUCT { ?X foaf:name ?FN . } WHERE { ?X VCard:FN ?FN . FILTER isLiteral(?FN) }
```

The filter expression here reduces the mapping by a kind of additional "type checking" where only those names are mapped which are merely given as a single literal.

*Example 2.* The situation becomes more tricky for other terms, for instance `VCard:n` (name) and `foaf:name`, because `VCard:n` consists of a substructure consisting of *Family name*, *Given name*, *Other names*, *honorific Prefixes*, and *honorific Suffixes*. One possibility is to concatenate all these to constitute a single `foaf:name`:

```
CONSTRUCT { ?X foaf:name ?Name . }
WHERE { ?X VCard:N ?N .
        OPTIONAL {?N VCard:Family ?Fam } OPTIONAL {?N VCard:Given  ?Giv }
        OPTIONAL {?N VCard:Other  ?Oth } OPTIONAL {?N VCard:Prefix ?Prefix }
        OPTIONAL {?N VCard:Suffix ?Suffix }
        FILTER (?Name = fn:concat(?Prefix," ",?Giv, " ",?Fam," ",?Oth," ",?Suffix))
      }
```

We observe the following problem here: First, we use filters for constructing a new binding which is not covered by the current SPARQL specification, since filter expressions are not meant to create new bindings of variables (in this case the variable

?Name), but only filter existing bindings. Second, if we wanted to model the case where e.g., several other names were provided, we would need built-in functions beyond what SPARQL currently provides, in this case a string manipulation function such as fn:concat. SPARQL supplies a subset of the functions and operators defined by XPath/XQuery, but these cover only boolean functions, like arithmetic comparison operators and basic arithmetic functions but no string manipulation routines. Even with the full range of XPath/XQuery functions available, we would still have to slightly "extend" fn:concat here, assuming that unbound variables are handled properly, being replaced by an empty string in case one of the optional parts of the name structure is not defined.

Apart from built-in functions like string operations, aggregate functions such as count, minimum, maximum or sum, are another helpful construct for many mappings that is currently not available in SPARQL. Finally, although we can query and create new RDF graphs by SPARQL CONSTRUCT statements mapping one vocabulary to another, there is no well-defined way to combine such mappings with arbitrary data, especially when we assume that (1) mappings are not restricted to be unidirectional from one vocabulary to another, but bidirectional, and (2) additional ontological inferences such as subclass/subproperty relations defined in the mutually mapped vocabularies should be taken into account when querying over syndicated RDF data and mappings. Hence, we propose the following extensions of SPARQL:

- We introduce an extensible set of useful built-in and aggregate functions.
- We permit function calls and aggregates in the CONSTRUCT clause,
- We further allow CONSTRUCT queries nested in FROM statements, or more general, allowing CONSTRUCT queries as part of the dataset.

### 2.1 Built-in Functions and Aggregates in Result Forms

Considering Example 2, it would be more intuitive to carry out the string translation from VCard:n to foaf:name in the result form, i.e., in the CONSTRUCT clause:

```
CONSTRUCT {?X foaf:name fn:concat(?Prefix," ",?Giv," ",?Fam," ",?Oth," ",?Suffix).}
WHERE { ?X VCard:N ?N .
        OPTIONAL {?N VCard:Family ?Fam } OPTIONAL {?N VCard:Given  ?Giv }
        OPTIONAL {?N VCard:Other  ?Oth } OPTIONAL {?N VCard:Prefix ?Prefix }
        OPTIONAL {?N VCard:Suffix ?Suffix } }
```

Another example for a non-trivial mapping is the different treatment of telephone numbers in FOAF and VCard.

*Example 3.* A VCard:tel is a foaf:phone – more precisely, VCard:tel is related to foaf:phone as follows. VCard stores Telephone numbers as string literals, whereas FOAF uses resources, i.e., URIs with the tel: URI-scheme:

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

Here we assumed the availability of a cast-function, which converts an xs:string to an RDF resource. While the distinction between literals and URI references in RDF usually makes perfect sense, this example shows that conversions between URI references and literals become necessary by practical uses of RDF vocabularies.

The next example shall illustrate the need for aggregate functions in mappings.

*Example 4.* The Description of a Project (DOAP) vocabulary[11] contains revision, i.e. version numbers of released versions of projects. With an aggregate function `MAX`, one can map DOAP information into the RDF Open Source Software Vocabulary [12], which talks about the latest release of a project, by picking the maximum value (numerically or lexicographically) of the set of revision numbers specified by a graph pattern as follows:

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

Here, the WHERE clause singles out all projects, while the aggregate selects the highest (i.e., latest) revision date of any available version for that project.

## 2.2 Nested CONSTRUCT Queries in FROM Clauses

The last example shows another example of "aggregation" which is not possible with SPARQL upfront, but may be realized by nesting CONSTRUCT queries in the FROM clause of a SPARQL query.

*Example 5.* Imagine you want to map/infer from an ontology having co-author relationships declared using `dc:creator` properties from the Dublin Core metadata vocabulary to `foaf:knows`, i.e., you want to specify "*If $?a$ and $?b$ have co-authored the same paper, then $?a$ knows $?b$*". The problem here is that a mapping using CONSTRUCT clauses needs to introduce new blank nodes for both $?a$ and $?b$ (since `dc:creator` is a datatype property usually just giving the name string of the author) and then need to infer the knows relation, so what we really want to express is a mapping

> If $?a$ and $?b$ are `dc:creator`s of the same paper, then someone named with `foaf:name` $?a$ `foaf:knows` someone with `foaf:name` $?b$.

A first-shot solution could be:

```
CONSTRUCT { _:a foaf:knows _:b . _:a foaf:name ?n1 . _:b foaf:name ?n2 . }
FROM <g>  WHERE { ?p dc:creator ?n1 . ?p dc:creator ?n2 . FILTER ( ?n1 != ?n2 ) }
```

Let us consider the present paper as example graph `g`:

```
g: <http://ex.org/papers#sparqlmappingpaper> dc:creator "Axel"
   <http://ex.org/papers#sparqlmappingpaper> dc:creator "Roman"
   <http://ex.org/papers#sparqlmappingpaper> dc:creator "Francois"
```

By the semantics of blank nodes in CONSTRUCT clauses — SPARQL creates new blank node identifiers for each solutions set matching the WHERE clause — the above would infer the following additional triples:

```
 _:a1 foaf:knows _:b1.  _:a1 foaf:name  "Axel".     _:b1 foaf:name  "Roman".
 _:a2 foaf:knows _:b2.  _:a2 foaf:name  "Axel".     _:b2 foaf:name  "Francois".
 _:a3 foaf:knows _:b3.  _:a3 foaf:name  "Francois". _:b3 foaf:name  "Roman".
 _:a4 foaf:knows _:b4.  _:a4 foaf:name  "Francois". _:b4 foaf:name  "Axel".
 _:a5 foaf:knows _:b5.  _:a5 foaf:name  "Roman".    _:b5 foaf:name  "Axel".
 _:a6 foaf:knows _:b6.  _:a6 foaf:name  "Roman".    _:b6 foaf:name  "Francois".
```

[11] `http://usefulinc.com/doap/`
[12] `http://xam.de/ns/os/`

Obviously, we lost some information in this mapping, namely the corellations that the "Axel" knowing "Francois" is the same "Axel" that knows "Roman", etc. We could remedy this situation by allowing to nest CONSTRUCT queries in the FROM clause of SPARQL queries as follows:

```
CONSTRUCT { ?a knows ?b . ?a foaf:name ?aname . ?b foaf:name ?bname . }
FROM { CONSTRUCT { _:auth foaf:name ?n . ?p aux:hasAuthor _:auth . }
       FROM <g> WHERE { ?p dc:creator ?n . } }
WHERE { ?p aux:hasAuthor ?a . ?a foaf:name ?aname .
        ?p aux:hasAuthor ?b . ?b foaf:name ?bname . FILTER ( ?a != ?b ) }
```

Here, the "inner" CONSTRUCT creates a graph with unique blank nodes for each author per paper, whereas the outer CONSTRUCT aggregates a more appropriate answer graph:

```
_:auth1 foaf:name "Axel". _:auth2 foaf:name "Roman". _:auth3 foaf:name "Francois".
_:auth1 foaf:knows _:auth2. _:auth1 foaf:knows _:auth3.
_:auth2 foaf:knows _:auth1. _:auth2 foaf:knows _:auth3.
_:auth3 foaf:knows _:auth1. _:auth3 foaf:knows _:auth2.
```

In Section 4, we will extend SPARQL to deal with these features. This extended version of the language, which we call SPARQL++ shall allow to evaluate SPARQL queries on top of RDF(S) data combined with mappings again expressed in SPARQL++.

## 3 Preliminaries – HEX-Programs

To evaluate SPARQL++ queries, we will translate them to so-called HEX-*programs* [**?**], an extension of logic programs under the answer-set semantics.

Let $Pred$, $Const$, $Var$, $exPr$ be mutually disjoint sets of predicate, constant, variable symbols, and external predicate names, respectively. In accordance with common notation in LP and the notation for external predicates from [9] we will in the following assume that $Const$ comprises the set of numeric constants, string constants beginning with a lower case letter, or double-quoted string literals, and IRIs.[13] $Var$ is the set of string constants beginning with an uppercase letter. Elements from $Const \cup Var$ are called *terms*. Given $p \in Pred$ an *atom* is defined as $p(t_1, \ldots, t_n)$, where $n$ is called the arity of $p$ and $t_1, \ldots, t_n$ are terms. An *external atom* is 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 $g \in exPr$ is an external predicate name. We assume the *input* 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.

[13] For the purpose of this paper, we will disregard language-tagged and datatyped literals in the translation to HEX-programs.

**Definition 1.** *A* rule *is of the form*

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

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

We use $H(r)$ to denote the head atom $h$ and $B(r)$ to denote the set of all body literals $B^+(r) \cup B^-(r)$ of $r$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$.

The notion of input and output terms in external atoms described above denotes the binding pattern. More precisely, we assume the following condition which extends the standard notion of safety (cf. [28]) in Datalog with negation.

**Definition 2 (Safety).** *Each variable appearing in a rule must appear in a non-negated body atom or as an output term of an external atom.*

Finally, we define HEX-programs.

**Definition 3.** *A* HEX-program *$P$ is defined as a set of safe rules $r$ of the form* (1).

The notions of *grounding*, *Herbrand Base* and *interpretation* correspond to traditional logic programming. With every external predicate name $e \in \mathit{exPr}$ we associate an $(n{+}m{+}1)$-ary Boolean function $f_e$ assigning each tuple $(I, y_1, \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n/m$ are the input/output arities of $e$, $I \subseteq \mathit{HB}_P$, $x_i \in \mathit{Const}$, and $y_j \in \mathit{Pred} \cup \mathit{Const}$. We say that $I \subseteq \mathit{HB}_P$ is a *model* of a ground external atom $a = e[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, iff $f_e(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.

Let $r$ be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$. We say that $I$ is a *model* of a HEX-program $P$, denoted $I \models P$, iff $I \models r$ for all $r \in \mathit{ground}(P)$.

The semantics we use here generalizes the answer-set semantics [13] and is defined using the *FLP-reduct* [12], which—contrary to the traditional Gelfond-Lifschitz reduct—ensures minimality of answer sets also in presence of external atoms: The *FLP-reduct* of $P$ with respect to $I \subseteq \mathit{HB}_P$, denoted $P^I$, is the set of all $r \in \mathit{ground}(P)$ such that $I \models B(r)$. $I \subseteq \mathit{HB}_P$ is an *answer set of* $P$ iff $I$ is a minimal model of $P^I$.

By the *cautious extension* of a predicate $p$ we denote the set of atoms with predicate symbol $p$ in the intersection of all answer sets of $P$.

For our purposes, we define a set of external predicates $\mathit{exPr} = \{\mathit{rdf}, \mathit{isBLANK}, \mathit{isIRI}, \mathit{isLITERAL}, =, \mathrel{!=}, \mathit{REGEX}, \mathit{CONCAT}, \mathit{COUNT}, \mathit{MAX}, \mathit{MIN}, \mathit{SK}\}$ with a fixed semantics as follows. These external predicates exemplarily demonstrate that HEX-programs are expressive enough to model all the necessary ingredients for evaluating SPARQL filters ($\mathit{isBLANK}, \mathit{isIRI}, \mathit{isLITERAL}, =, \mathrel{!=}, \mathit{REGEX}$) and also for more expressive built-in functions and aggregates ($\mathit{CONCAT}, \mathit{SK}, \mathit{COUNT}, \mathit{MAX}, \mathit{MIN}$). Here, $\mathit{CONCAT}$ is just an example built-in, assuming that more XPath/XQuery functions could similarly be added.

For the $\mathit{rdf}$ predicate we write atoms as $\mathit{rdf}[i](s,p,o)$ with an input term $i \in \mathit{Const} \cup \mathit{Var}$ and output terms $s, p, o \in \mathit{Const}$. The external atom $\mathit{rdf}[i](s,p,o)$ is true if $(s,p,o)$ is an RDF triple *entailed* by the RDF graph at IRI $i$. For the moment, here we consider simple RDF entailment [15] only.

The atoms $isBLANK[c](val)$, $isIRI[c](val)$, $isLITERAL[c](val)$ test input term $c \in Const \cup Var$ for being a valid string representation of a blank node, IRI reference or RDF literal. The atom $REGEX[c_1, c_2](val)$ test whether $c_1$ matches the regular expression $c_2$. All these external predicates return an output value $val \in \{\mathtt{t}, \mathtt{f}, \mathtt{e}\}$, representing truth, falsity or an error, following the semantics defined in [24, Sec. 11.3].

Apart from these truth-valued external atoms we add other external predicates which mimic built-in functions an aggregates. As an example predicate for a built-in, we chose the predicate $CONCAT[c_1, \ldots, c_n](c_{n+1})$ with variable input arity which concatenates string constants $c_1, \ldots, c_n$ into $c_{n+1}$ and thus implements the semantics of `fn:concat` in XPath/XQuery [19].

Next, we define external predicates which mimic aggregate functions over a certain predicate. Let $p \in Pred$ with arity $n$, and $x_1, \ldots, x_n \in Const \cup \{mask\}$ where $mask$ is a special constant not allowed to appear anywhere except in input lists of aggregate predicates. Then $COUNT[p, x_1, \ldots, x_n](c)$ is true if $c$ equals the number of distinct tuples $(t_1, \ldots, t_n)$, such that $I \models p(t_1, \ldots, t_n)$ and for all $x_i$ different from the constant $mask$ it holds that $t_i = x_i$. $MAX[p, x_1, \ldots, x_n](c)$ (and $MIN[p, x_1, \ldots, x_n](c)$, resp.) is true if among all tuples $(t_1, \ldots, t_n)$, such that $I \models p(t_1, \ldots, t_n)$, $c$ is the lexicographically greatest (smallest, resp.) value among all the $t_i$ such that $x_i = mask$.[14]

We will illustrate the use of these external predicates to express aggregations below when discussing the actual translation from SPARQL++ to HEX-programs.

Finally, the external predicate $SK[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$ from its input parameters. We will use this built-in function in our translation of SPARQL queries with blank nodes in the CONSTRUCT part. Similar to the aggregate functions mentioned before, when using $SK$ we will need to take special care in our translation in order to retain strong safety.

As widely known, for programs without external predicates, safety guarantees that the number of entailed ground atoms is finite. Though, by external atoms in rule bodies, new, possibly infinitly many, ground atoms could be generated, even if all atoms themselves are safe. In order to avoid this, a stronger notion of safety for HEX-programs is defined in [27]: Informally, this notion says that a HEX-program is *strongly safe*, if no external predicate recursively depends on itself, thus defining a notion of stratification over external predicates. Strong safety guarantees finiteness of models as well as finite computability of external atoms.

## 4 Extending SPARQL towards mappings

In Section 2 we have shown that an extension of the CONSTRUCT clause is needed for SPARQL to be suitable for mapping tasks. In the following, we will formally define extended SPARQL queries which allow to integrate built-in functions and aggregates in CONSTRUCT clauses as well as in FILTER expressions. We will define the semantics of such extended queries, and, moreover, we will provide a translation to HEX-programs, building upon an existing translation presented in [22].

[14] Note that in this definition we allow min/max to aggregate over several variables.

A SPARQL++ *query* $Q = (R, P, DS)$ consists of a result form $R$, a graph pattern $P$, and an extended dataset $DS$ as defined below.[15] We refer to [24] for syntactical details and will explain these in the following as far as necessary.

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, and variables respectively. $I \cup L \cup Var$ is also called the set of *basic RDF terms*. In this paper, we allow as blank node identifiers nested ground terms similar to HiLog terms [4], such that $B$ is defined recursively over an infinite set of constant blank node identifiers $B_c$ as follows:

– each element of $B_c$ is a blank node identifier, i.e., $B_c \subseteq B$.
– for $b \in B$ and $t_1, \ldots, t_n$ in $I \cup B \cup L$, $b(t_1, \ldots, t_n) \in B$.

Now, we extend the SPARQL syntax by allowing built-in functions and aggregates in place of basic RDF terms in graph patterns (and thus also in CONSTRUCT clauses) as well as in filter expressions. We define the set *Blt* of *built-in terms* as follows:

– All basic terms are built-in terms.
– If *blt* is a built-in predicate (e.g., `fn:concat` from above or another XPath/XQuery functions), and $c_1, \ldots, c_n$ are built-in terms then $blt(c_1, \ldots, c_n)$ is a built-in term.
– If $agg$ is an aggregate function (e.g., $COUNT$, $MIN$, $MAX$), $P$ a graph pattern, and $V$ a tuple of variables appearing in $P$, then $agg\,(V : P)$ is a built-in term.[16]

In the following we will introduce extended graph patterns that may include built-in terms and extended datasets that can be constituted by CONSTRUCT queries.

## 4.1 Extended Graph Patterns

As for *graph patterns*, we follow the recursive definition from [21]:

– A triple pattern $(s, p, o)$ is a graph pattern where $s, o \in Blt$ and $p \in I \cup Var$.[17] Triple patterns which only contain basic terms are called *basic triple patterns* and *value-generating triple patterns* otherwise.
– Let $P_1, P_2$ be graph patterns, $i \in I \cup Var$, $R$ a filter expression, then ($P_1$ FILTER $R$), ($P_1$ OPT $P_2$), ($P_1$ UNION $P_2$), (GRAPH $i$ $P_1$), and ($P_1$ AND $P_2$) are graph patterns.[18]

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

[15] As we deal mainly with CONSTRUCT queries here, we will ignore solution modifiers.
[16] This aggregate syntax is adapted from the resp. definition for aggregates in LP from [12].
[17] We do not consider blanks nodes here as these can be equivalently replaced by variables [5].
[18] We use AND to keep with the operator style of [21] although it is not explicit in SPARQL.

Similar to aggregates in logic programming, we use a notion of safety. First, given a query $Q = (R, P, DS)$ we allow only basic triple patterns in $P$, ie. we only allow built-ins and aggregates only in FILTERs or in the result pattern $R$. Second, a built-in term $blt$ occurring in the result form or in $P$ in a query $Q = (R, P, DS)$ is *safe* if all variables recursively appearing in $blt$ also appear in a basic triple pattern within $P$ .

## 4.2 Extended Datasets

In order to allow the definition of RDF data side-by-side with implicit data defined by mappings of different vocabularies or, more general, views within RDF, we define an *extended RDF graph* as a set of RDF triples $I \cup L \cup B \ \times \ I \ \times \ I \cup L \cup B$ and CONSTRUCT queries. An RDF graph (or dataset, resp.) without CONSTRUCT queries is called a *basic graph* (or dataset, resp.).

The dataset $DS = (G, \{(g_1, G_1), \dots (g_k, G_k)\})$ of a SPARQL query is defined by (i) a default graph $G$, i.e., the RDF merge [15, Section 0.3] of a set of extended RDF graphs, plus (ii) a set of named graphs, i.e., pairs of IRIs and corresponding extended graphs. Without loss of generality (there are other ways to define the dataset such as in a SPARQL protocol query), we assume $DS$ defined by the IRIs given in a set of FROM and FROM NAMED clauses. As an exception, we assume that any CONSTRUCT query which is part of an extended graph $G$ by default (i.e., in the absence of FROM and FROM NAMED clauses) has the dataset $DS = (G, \emptyset)$ For convenience, we allow extended graphs consisting of a single CONSTRUCT statement to be written directly in the FROM clause of a SPARQL++ query, like in Example 5.

We will now define syntactic restrictions on the CONSTRUCT queries allowed in extended datasets, which retain finite termination on queries over such datasets. Let $G$ be an extended graph. First, for any CONSTRUCT query $Q = (R, P, DS_Q)$ in $G$, $DS_Q$ we allow only triple patterns $tr = (s, p, o)$ in $P$ or $R$ where $p \in I$, i.e., neither blank nodes nor variables are allowed in predicate positions in extended graphs, and, additionally, $o \in I$ for all triples such that $p = \texttt{rdf:type}$. Second, we define a predicate-class-dependency graph over an extended dataset $DS = (G, \{(g_1, G_1), \dots (g_k, G_k)\})$ as follows. The predicate-class-dependency graph for $DS$ has an edge $p \rightarrow r$ with $p, r \in I$ for any CONSTRUCT query $Q = (R, P, DS)$ in $G$ with $r$ (or $p$, resp.) either (i) a predicate different from `rdf:type` in a triple in $R$ (or $P$, resp.), or (ii) an object in an `rdf:type` triple in $R$ (or $P$, resp.). All edges such that $r$ occurs in a value-generating triple are marked with '$*$'. We now say that $DS$ is *strongly safe* if its predicate-class-dependency graph does not contain any cycles involving marked edges. As it turns out, in our translation in Section 4.4 below, this condition is sufficient (but not necessary) to guarantee that any query can be translated to a strongly safe HEX-program.

Like in [26] we assume that blank node identifiers in each query $Q = (R, P, DS)$ have been standardized apart, i.e., that no blank nodes with the same identifiers appear in a different scope. The scope of a blank node identifier is defined as the graph or graph pattern it appears in, where each WHERE or CONSTRUCT clause open a "fresh" scope . For instance, take the extended graph dataset in Fig. 1(a), its standardized apart version is shown in Fig. 1(b). Obviously, extended datasets can always be standardized apart in linear time in a preprocessing step.

```
g1:  :paper2 foaf:maker _:a.
     _:a foaf:name "Jean Deau".

g2: :paper1 dc:creator "John Doe".
    :paper1 dc:creator "Joan Dough".
    CONSTRUCT {_:a foaf:knows _:b .
               _:a foaf:name ?N1 .
               _:b foaf:name ?N2 . }
    WHERE {?X dc:creator ?N1,?N2.
           FILTER( ?N1 != ?N2 ) }
```

(a)

```
g1:  :paper2 foaf:maker _:b1.
     _:b1 foaf:name "Jean Deau".

g2: :paper1 dc:creator "John Doe".
    :paper1 dc:creator "Joan Dough".
    CONSTRUCT {_:b2 foaf:knows _:b3 .
               _:b2 foaf:name ?N1 .
               _:b3 foaf:name ?N2 . }
    WHERE {?X dc:creator ?N1,?N2.
           FILTER( ?N1 != ?N2 ) }
```

(b)

**Fig. 1.** Standardizing apart blank node identifiers in extended datasets.

## 4.3 Semantics

The semantics of SPARQL++ is based on the formal semantics for SPARQL by Pérez et al. in [21] and its translation into HEX-programs in [22]. We denote by $T_{\mathsf{null}}$ the union $I \cup B \cup L \cup \{\mathsf{null}\}$, where null is a dedicated constant denoting the unknown value not appearing in any of $I$, $B$, or $L$, how it is commonly introduced when defining outer joins in relational database systems. A *substitution* $\theta$ from $Var$ to $T_{\mathsf{null}}$ is a partial function $\theta : Var \rightarrow T_{\mathsf{null}}$. We write substitutions in postfix notation in square brackets, i.e., if $t, t' \in Blt$ and $v \in Var$, then $t[v/t']$ is the term obtained from replacing all occcurences of $v$ in $t$ by $t'$. The *domain* of $\theta$, $dom(\theta)$, is the subset of $Var$ where $\theta$ is defined. The lexicographic ordering of this subset is denoted by $\overline{dom}(Var)$. For a substitution $\theta$ and a set of variables $D \subseteq Var$ we define the substitution $\theta^D$ with domain $D$ as follows

$$x\theta^D = \begin{cases} x\theta & \text{if } x \in dom(\theta) \cap D \\ \mathsf{null} & \text{if } x \in D \setminus dom(\theta) \end{cases}$$

Let $x \in Var$, $\theta_1, \theta_2$ be substitutions, then $\theta_1 \cup \theta_2$ is the substitution obtained as follows:

$$x(\theta_1 \cup \theta_2) = \begin{cases} \phantom{\text{else: }} x\theta_1 & \text{if } x\theta_1 \text{ defined and } x\theta_2 \text{ undefined} \\ \text{else: } x\theta_1 & \text{if } x\theta_1 \text{ defined and } x\theta_2 = \mathsf{null} \\ \text{else: } x\theta_2 & \text{if } x\theta_2 \text{ defined} \\ \text{else: undefined} & \end{cases}$$

Thus, in the union of two substitutions defined values in one take precedence over null values the other substitution. Two substitutions $\theta_1$ and $\theta_2$ are *compatible* when for all $x \in dom(\theta_1) \cap dom(\theta_2)$ either $x\theta_1 = \mathsf{null}$ or $x\theta_2 = \mathsf{null}$ or $x\theta_1 = x\theta_2$ holds, i.e., when $\theta_1 \cup \theta_2$ is a substitution over $dom(\theta_1) \cup dom(\theta_2)$. Analogously to Pérez et al. we define join, union, difference, and outer join between two sets of substitutions $\Omega_1$ and $\Omega_2$ over domains $D_1$ and $D_2$, respectively:

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 &= \{\theta_1 \cup \theta_2 \mid \theta_1 \in \Omega_1, \theta_2 \in \Omega_2, \theta_1 \text{ and } \theta_2 \text{ are compatible}\} \\ \Omega_1 \cup \Omega_2 &= \{\theta \mid \exists \theta_1 \in \Omega_1 \text{ with } \theta = \theta_1^{D_1 \cup D_2} \text{ or } \exists \theta_2 \in \Omega_2 \text{ with } \theta = \theta_2^{D_1 \cup D_2}\} \\ \Omega_1 - \Omega_2 &= \{\theta \in \Omega_1 \mid \text{ for all } \theta_2 \in \Omega_2,\ \theta \text{ and } \theta_2 \text{ not compatible}\} \\ \Omega_1 \mathbin{\sqsupset\!\!\bowtie} \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2) \end{aligned}$$

Next, we define the application of substitutions to built-in terms and triples: For a built-in term $t$, by $t\theta$ we denote the value obtained by applying the substitution to all variables in $t$. By $eval_\theta(t)$ we denote the value obtained by (i) recursively evaluating all built-in and aggregate functions, and (ii) replacing all bNode identifiers by complex bNode identifiers according to $\theta$, as follows:

| | |
|---|---|
| $eval_\theta(\texttt{fn:concat}(c_1, c_2, \ldots, c_n))$ | Returns the `xs:string` that is the concatenation of the values of $c_1\theta$,...,$c_1\theta$ after conversion. If any of the arguments is the empty sequence or null, the argument is treated as the zero-length string. |
| $eval_\theta(\texttt{COUNT}(V : P))$ | Returns the number of distinct[19] answer substitutions for the query $Q = (V, P\theta, DS)$ where $DS$ is the dataset of the encapsulating query. |
| $eval_\theta(\texttt{MAX}(V : P))$ | Returns the maximum (numerically or lexicographically) of distinct answer substitutions for the query $Q = (V, P\theta, DS)$. |
| $eval_\theta(\texttt{MIN}(V : P))$ | Analogous to `MAX`, but returns the minimum. |
| $eval_\theta(t)$ | Returns $t\theta$ for all $t \in I \cup L \cup Var$, and $t(\overline{dom}(\theta)\theta)$ for $t \in B$.[20] |

Finally, for a triple pattern $tr = (s, p, o)$ we denote by $tr\theta$ the triple $(s\theta, p\theta, o\theta)$, and by $eval_\theta(tr)$ the triple $(eval_\theta(s), eval_\theta(p), eval_\theta(o))$.

The evaluation of a graph pattern $P$ over a basic dataset $DS = (G, G_n)$, can now be defined recursively by sets of substitutions, extending the definitions in [21, 22].

**Definition 4.** *Let $tr = (s, p, o)$ be a basic triple pattern, $P, P_1, P_2$ graph patterns, and $DS = (G, G_n)$ a basic dataset, then the* evaluation $[\![\cdot]\!]_{DS}$ *is defined as follows:*

$$
\begin{array}{ll}
[\![tr]\!]_{DS} = & \{\theta \mid dom(\theta) = vars(P) \text{ and } tr\theta \in G\} \\
[\![P_1 \textsf{ AND } P_2]\!]_{DS} & = [\![P_1]\!]_{DS} \bowtie [\![P_2]\!]_{DS} \\
[\![P_1 \textsf{ UNION } P_2]\!]_{DS} & = [\![P_1]\!]_{DS} \cup [\![P_2]\!]_{DS} \\
[\![P_1 \textsf{ OPT } P_2]\!]_{DS} & = [\![P_1]\!]_{DS} ⟕ [\![P_2]\!]_{DS} \\
[\![\textsf{GRAPH } i\ P]\!]_{DS} & = [\![P]\!]_{(i,\emptyset)}, \text{for } i \in G_n \\
[\![\textsf{GRAPH } v\ P]\!]_{DS} & = \{\theta \cup [v/g] \mid g \in G_n \text{ and } \theta \in [\![P[v/g]\,]\!]_{(g,\emptyset)}\}, \text{for } v \in Var \\
[\![P \textsf{ FILTER } R]\!]_{DS} & = \{\theta \in [\![P]\!]_{DS} \mid R\theta = \top\}
\end{array}
$$

*Let $R$ be a filter expression, $u, v \in Blt$. The valuation of $R$ on a substitution $\theta$, written $R\theta$ takes one of the three values $\{\top, \bot, \varepsilon\}$[21] and is defined as follows.*

$R\theta = \top$, *if:* (1) $R = \textsf{BOUND}(v)$ *with* $v \in dom(\theta) \wedge eval_\theta(v) \neq \textsf{null}$;
(2) $R = \textsf{isBLANK}(v)$ *with* $eval_\theta(v) \in B$;
(3) $R = \textsf{isIRI}(v)$ *with* $eval_\theta(v) \in I$;
(4) $R = \textsf{isLITERAL}(v)$ *with* $eval_\theta(v) \in L$;
(5) $R = (u = v)$ *with* $eval_\theta(u) = eval_\theta(v) \wedge eval_\theta(u) \neq \textsf{null}$;
(6) $R = (\neg R_1)$ *with* $R_1\theta = \bot$;
(7) $R = (R1 \vee R2)$ *with* $R_1\theta = \top \vee R_2\theta = \top$;
(8) $R = (R1 \wedge R2)$ *with* $R_1\theta = \top \wedge R_2\theta = \top$.

$R\theta = \varepsilon$, *if:* (1) $R = \textsf{isBLANK}(v), R = \textsf{isIRI}(v), R = \textsf{isLITERAL}(v)$, *or*
$R = (u = v)$ *with* $(v \in Var \wedge v \notin dom(\theta)) \vee eval_\theta(v) = \textsf{null} \vee$
$(u \in Var \wedge u \notin dom(\theta)) \vee eval_\theta(u) = \textsf{null}$;
(2) $R = (\neg R_1)$ *and* $R_1\theta = \varepsilon$;
(3) $R = (R_1 \vee R_2)$ *and* $R_1\theta \neq \top \wedge R_2\theta \neq \top \wedge (R_1\theta = \varepsilon \vee R_2\theta = \varepsilon)$;
(4) $R = (R1 \wedge R2)$ *and* $R_1\theta = \varepsilon \vee R_2\theta = \varepsilon$.

$R\theta = \bot$ *otherwise.*

In [22] we have shown that the semantics defined this way corresponds with the original semantics for SPARQL defined in [21] without complex built-in and aggregate terms and on basic datasets.[22]

[19] Note that we give a set based semantics to the counting built-in, we do not take into account duplicate solutions which can arise from the multi-set semantics in [24] when counting.

[20] For blank nodes $eval_\theta$ constructs a new blank node identifier, similar to Skolemization.

[22] Our definition here only differs in in the application of $eval_\theta$ on built-in terms in filter expressions which does not make a difference if only basic terms appear in FILTERs.

Note that, so far we have only defined the semantics in terms of a pattern $P$ and basic dataset $DS$, but neither taken the result form $R$ nor extended datasets into account. As for the former, we proceed with formally define solutions for SELECT and CONSTRUCT queries, respectively. The semantics of a SELECT query $Q = (V, P, DS)$ is fully determined by its *solution tuples* [22].

**Definition 5.** *Let $Q = (R, P, DS)$ be a* SPARQL++ *query, and $\theta$ a substitution in $[\![P]\!]_{DS}$, then we call the tuple $\overline{vars(P)}\theta$ a solution tuple of $Q$.*

As for a CONSTRUCT queries, we define the *solution graphs* as follows.

**Definition 6.** *Let $Q = (R, P, DS)$ be a* SPARQL CONSTRUCT *query where blank node identifiers in DS and R have been standardized apart and $R = \{t_1, \ldots, t_n\}$ is the result graph pattern. Further, for any $\theta \in [\![P]\!]_{DS}$, let $\theta' = \theta^{vars(R) \cup vars(P)}$. The solution graph for Q is then defined as the triples obtained from*

$$\bigcup_{\theta in [\![P]\!]_{DS}} \{eval_{\theta'}(t_1), \ldots, eval_{\theta'}(t_n)\}$$

*by eliminating all non-valid RDF triples.*[23]

Our definitions so far only cover basic datasets. Extended datasets, which are implicitly defined bring the following additional challenges: (i) it is not clear upfront which blank node identifiers to give to blank nodes resulting from evaluating CONSTRUCT clauses, and (ii) extended datasets might involve recursive CONSTRUCT definitions which construct new triples in terms of the same graph in which they are defined. As for (i), we remedy the situation by constructing new identifier names via a kind of Skolemization, as defined in the function $eval_\theta$, see the table on page 12. $eval_\theta$ generates a unique blank node identifier for each solution $\theta$. Regarding (ii) we avoid possibly infinite datasets over recursive CONSTRUCT clauses by the strong safety restriction in Section 4.2. Thus, we can define a translation from extended datasets to HEX-programs which uniquely identifies the solutions for queries over extended datasets.

## 4.4 Translation to HEX-Programs

Our translation from SPARQL++ queries to HEX-programs is based on the translation for non-extended SPARQL queries outlined in [22]. Similar to the well-known correspondence between SQL and Datalog, SPARQL++ queries can be expressed by HEX-programs, which provide the additional machinery necessary for importing and processing RDF data as well as evaluating built-ins and aggregates. The translation consists of two basic parts: (i) rules that represent the query's graph pattern (ii) rules defining the triples in the extended datasets.

We have shown in [22] that solution tuples for any query $Q$ can be generated by a logic program and are represented by the extension of a designated predicate $\texttt{answer}_Q$, assuming that the triples of the dataset are available in a predicate $\texttt{triple}_Q$. We refer to [22] for details and only outline the translation here by examples.

[23] That is, triples with null values or blank nodes in predicate position, etc.

Complex graph patterns can be translated recursively in a rather straightforward way, where unions and join of graph patterns can directly be expressed by appropriate rule constructions, whereas OPTIONAL patterns involve negation as failure.

*Example 6.* Let query $q$ select all persons who do not know anybody called "John Doe" from the extended dataset $DS = (g1 \cup g2, \emptyset)$, i.e., the merge of the graphs in Fig. 1(b).

```
SELECT  ?P FROM <g1> FROM <g2>
WHERE { ?P rdf:type foaf:Agent . FILTER ( !BOUND(?P1) )
        OPTIONAL { P? foaf:knows ?P1 . ?P1 foaf:name "John Doe" . } }
```

This query can be translated to the following HEX-program:

```
answer_q(P) :- answer1_q(P,P1), P1 = null.
answer1_q(P,P1) :- answer2_q(P), answer_q3(P,P1).
answer1_q(P,null) :- answer2_q(P), not answer3_q'(P).
answer2_q(P) :- triple_q(P,rdf:type,foaf:Agent,def).
answer3_q(P,P1) :- triple_q(P,foaf:knows,P1,def),triple(P1,foaf:name,"John Doe",def).
answer3_q'(P) :- answer3_q(P,P1).
```

More complex queries with nested patterns can be translated likewise by introducing more auxiliary predicates. The program part defining the $\texttt{triple}_q$ predicate fixes the triples of the dataset, by importing all explicit triples in the dataset as well as recursively translating all CONSTRUCT clauses and subqueries in the extended dataset.

*Example 7.* The program to generate the dataset triples for the extended dataset $DS = (g1 \cup g2, \emptyset)$ looks as follows:

```
triple_q(S,P,O,def) :- rdf["g1"](S,P,O).
triple_q(S,P,O,def) :- rdf["g2"](S,P,O).
triple_q(B2,foaf:knows,B3,def) :- SK[b2(X,N1,N2)](B2),SK[b3(X,N1,N2)](B3),
                                  answer_{C1,g2}(X,N1,N2).
triple_q(B2,foaf:name,N1,def) :- SK[b2(X,N1,N2)](B2), answer_{C1,g2}(X,N1,N2).
triple_q(B3,foaf:knows,N2,def) :- SK[b3(X,N1,N2)](B3), answer_{C1,g2}(X,N1,N2).
answer_{C1,g2}(X,N1,N2) :- triple_q(X,dc:creator, N1,def),
                           tripleq(X,dc:creator,N2,def), N1 != N2.
```

The first two rules import all triples given explicitly in graphs $g1, g2$ by means of the "standard" RDF import HEX predicate. The next three rules create the triples from the CONSTRUCT in graph $g2$, where the query pattern is translated by an own subprogram defining the predicate $\texttt{answer}_{C_1,g2}$, which in this case only consists of a single rule.

The example shows the use of the external function $SK$ to create blank node ids for each solution tuple as mentioned before, which we need to emulate the semantics of blank nodes in CONSTRUCT statements.

Next, we turn to the use of HEX aggregate predicates in order to translate aggregate terms. Let $Q = (R, P, DS)$ and $a = agg\,(V : P_a)$ – here, $V \subseteq vars(P_a)$ is the tuple of variables we want to aggregate over – be an aggregate term appearing either in $R$ or in a filter expression in $P$. Then, the idea is that $a$ can be translated by an external atom $agg[aux, \overline{vars}(P_a)'[V/mask]\,](v_a)$ where

(i) $\overline{vars}(P_a)'$ is obtained from $\overline{vars}(P_a)$ by removing all variables which are not aggregated over and only appear in $P_a$ but not elsewhere in $P$, i.e., from $vars(P_a) \cap vars(P) \cup V$
(ii) the variable $v_a$ takes the place of $a$,
(iii) $aux_a$ is a new predicate defined by a rule: $aux_a(\overline{vars}(P_a)') \leftarrow answer_a(\overline{vars}(P_a))$.
(iv) $answer_a$ is the predicate defining the solution set of the query $Q_a = (vars(P_a), P_a, DS)$

*Example 8.* The following rules mimic the CONSTRUCT query of Example 4:

```
triple(P,os:latestRelease,V$_a$) :- MAX[aux$_a$,P,R,mask](V$_a$),
                                triple(P,rdf:type,doap:Project,gr).
aux$_a$(P,R,V) :- answer$_a$(P,R,V).
answer$_a$(P,R,V) :- triple(P,doap:release R,def), triple(R,doap:revision,V,def).
```

With the extensions the translation in [22] outlined here for extended datasets, aggregate and built-in terms we can define the solution tuples of an SPARQL++ query $Q = (R, P, DS)$ over an extended dataset now as precisely the set of tuples corresponding to the cautious extension of the predicate answer$_q$.

## 4.5 Adding ontological inferences by encoding $\rho$df$^-$ into SPARQL

Trying the translation sketched above on the query in Example 6 we observe that we would not obtain any answers, as no triples in the dataset would match the triple pattern `?P rdf:type foaf:Agent` in the WHERE clause. This still holds if we include the vocabulary definition of FOAF at `http://xmlns.com/foaf/spec/index.rdf` to the dataset, since the machinery introduced so far could not draw any additional inferences from the triple `foaf:maker rdfs:range foaf:Agent` which would be necessary in order to figure out that Jean Deau is indeed an agent. There are several open issues on using SPARQL on higher entailment regimes than simple RDF entailment which allow such inferences. One such problem is the presences of infinite axiomatic triples in RDF semantics or several open compatibility issues with OWL semantics, see also [9]. However, we would like to at least add some of the inferences of the RDFS semantics. To this end, we will encode a small but very useful subset of RDFS, called $\rho$df [20] into the extended dataset. $\rho$df, defined by Muñoz et al., restricts the RDF vocabulary to its essentials by only focusing on the properties rdfs:subPropertyOf, rdfs:subClassOf,rdf:type, rdfs:domain, and rdfs:range, ignoring other constituents of the RDFS vocabulary. Most importantly, Muñoz et al. prove that (i) $\rho$df entailment corresponds to full RDF entailment on graphs not mentioning RDFS vocabulary outside $\rho$df, and (ii) that $\rho$df entailment can be reduced to five axiomatic triples (concerned with reflexivity of the subproperty relationship) and 14 entailment rules. Note that for graphs which do not mention subclass or subproperty relationships, which is usually the case for patterns in SPARQL queries or the mapping rules we encode here, even a reflexive-relaxed version of $\rho$df that does not contain any axiomatic triples is sufficient. We can write down all but one of the entailment rules of reflexive-relaxed $\rho$df as CONSTRUCT queries which we consider implicitly present in the extended dataset:

```
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.}
```

There is one more entailment rule for reflexive-relaxed $\rho$df concerning that blank node renaming preserves $\rho$df entailment. However, it is neither straightforwardly possible, nor desirable to encode this by CONSTRUCTs like the other rules. Blank node renaming might have unintuitive effects on aggregations and in connection with OPTIONAL queries. In fact, keeping blank node identifiers in recursive CONSTRUCTs after standardizing apart is what keeps our semantics finite, so we skip this rule, and call the resulting $\rho$df fragment encoded by the above CONSTRUCTs $\rho\text{df}^{-}$. Some care is in order concerning strong safety of the resulting dataset when adding $\rho\text{df}^{-}$. To still ensure strong safety of the translation, we complete the predicate-class-dependency graph by additional edges between all pairs of resources connected by subclassOf or subPropertyOf, domain, or range relations and checking the same safety condition as before on the graph extended in this manner.

### 4.6 Implementation

We implemented a prototype of a SPARQL++ engine based on on the HEX-program solver dlvhex.[24] 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 Section 4.4. The system implements external atoms used in the translation, namely (i) the RDF atom for data import, (ii) the aggregate atoms, and (iii) a string concatenation atom implementing both the $CONCAT$ function and the $SK$ atom for bNode handling. The engine can directly be fed with a SPARQL++ query. 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, which can subsequently be transformed easily to a valid RDF syntax by an XSLT to export solution graphs.

## 5 Related work

The idea of using SPARQL CONSTRUCT queries is in fact not new, even some implemented systems such as TopBraid Composer already seem to offer this feature, [25] however without a defined and layered semantics, and lacking aggregates or built-ins, thus insufficient to express mappings such as the ones studied in this article.

Our notion of extended graphs and datasets generalizes so-called networked graphs defined by Schenk and Staab [26] who also use SPARQL CONSTRUCT statements as rules with a slightly different motivation: dynamically generating views over graphs. The authors only permit bNode- and built-in free CONSTRUCTs whereas we additionally allow bNodes, built-ins and aggregates, as long as strong safety holds which only restricts recursion over value-generating triples. Another differenece is that their semantics bases on the well-founded instead of the stable model semantics.

PSPARQL [1], a recent extension of SPARQL, allows to query RDF graphs using regular path expressions over predicates. This extension is certainly useful to represent mappings and queries over graphs. We conjecture that we can partly emulate such path expressions by recursive CONSTRUCTs in extended datasets.

As an interesting orthogonal approach, we mention iSPARQL [17] which proposes an alternative way to add external function calls to SPARQL by introducing so called

[24] Available with dlvhex on `http://www.kr.tuwien.ac.at/research/dlvhex/`.

[25] `http://composing-the-semantic-web.blogspot.com/2006/09/ontology-mapping-with-sparql-construct.html`

virtual triple patterns which query a “virtual” dataset that could be an arbitrary service. This approach does not need syntactic extensions of the language. However, an implementation of this extension makes it necessary to know upfront which predicates denote virtual triples. The authors use their framework to call a library of similarity measure functions but do not focus on mappings or CONSTRUCT queries.

As already mentioned in the introduction, other approaches often allow only mappings at the level of the ontology level or deploy their own rules language such as SWRL [16] or WRL [7]. A language more specific for ontology mapping is C-OWL [3], which extends OWL with bridge rules to relate ontological entities. C-OWL is a formalism close to distributed description logics [2]. These approaches partially cover aspects which we cannot handle, e.g., equating instances using owl:sameAs in SWRL or relating ontologies based on a local model semantics [14] in C-OWL. None of these approaches though offers aggregations which are often useful in practical applications of RDF data syndication, the main application we target in the present work. The Ontology Alignment Format [10] and the Ontology Mapping Language [25] are ongoing efforts to express ontology mappings. In a recent work [11], these two languages were merged and given a model-theoretic semantics which can be grounded to a particular logical formalism in order to be actually used to perform a mediation task. Our approach combines rule and mapping specification languages using a more practical approach than the above mentioned, exploiting standard languages, $\rho$df and SPARQL. We keep the ontology language expressivity low on purpose in order to retain decidability, thus providing an executable mapping specification format.

As a final remark, let us emphasize that our translation is based on the set-based semantics of [21, 22] whereas the algebra for SPARQL defined in the latest candidate recommendation [24] defines a multiset semantics. An extension of our translation towards this multiset semantics is described in [23].

## 6 Conclusions and Further Work

In this paper we have demonstrated the use of SPARQL++ as a rule language for defining mappings between RDF vocabularies, allowing CONSTRUCT queries — extended with built-in and aggregate functions — as part of the dataset of SPARQL queries. We mainly aimed at setting the theoretical foundations for SPARQL++. Our next steps will involve to focus on scalability of our current prototype, by looking into how far evaluation of SPARQL++ queries can be optimized, for instance, by pushing query evaluation from our dlvhex as far as possible into more efficient SPARQL engines or possibly distributed SPARQL endpoints that cannot deal with extended datasets natively. Further, we will investigate the feasibility of supporting larger fragments of RDFS and OWL. Here, caution is in order as arbitrary combininations of OWL and SPARQL++ involve the same problems as combining rules with ontologies (see[9]) in the general case. We believe that the small fragment we started with is the right strategy in order to allow queries over networks of lightweight RDFS ontologies, connectable via expressive mappings, which we will gradually extend.

## References

1. F. Alkhateeb, J.-F. Baget, J. Euzenat. Extending SPARQL with Regular Expression Patterns. Tech. Report 6191, Inst. National de Recherche en Informatique et Automatique, May 2007.
2. A. Borgida, L. Serafini. Distributed Description Logics: Assimilating Information from Peer Sources. *Journal of Data Semantics*, 1:153–184, 2003.
3. P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, H. Stuckenschmidt. C-OWL: Contextualizing Ontologies. In *The Semantic Web - ISWC 2003*, Florida, USA, 2003.
4. W. Chen, M. Kifer, D. Warren. HiLog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
5. J. de Bruijn, E. Franconi, S. Tessaris. Logical Reconstruction of Normative RDF. In *OWL: Experiences and Directions Workshop (OWLED-2005)*, Galway, Ireland, 2005.
6. J. de Bruijn, S. Heymans. A Semantic Framework for Language Layering in WSML. In *First Int'l Conf. on Web Reasoning and Rule Systems (RR2007)*, Innsbruck, Austria, 2007.
7. J. de Bruijn (ed.). Web Rule Language (WRL), 2005. W3C Member Submission.
8. S. Decker et al. TRIPLE - an RDF Rule Language with Context and Use Cases. In *W3C Workshop on Rule Languages for Interoperability*, Washington D.C., USA, April 2005.
9. T. Eiter, G. Ianni, A. Polleres, R. Schindlauer, H. Tompits. Reasoning with Rules and Ontologies. In *Reasoning Web 2006*, pp. 93–127. Springer, Sept. 2006.
10. J. Euzenat. An API for Ontology Alignment. In *Proc. 3rd International Semantic Web Conference, Hiroshima, Japan*, pp. 698–712, 2004.
11. J. Euzenat, F. Scharffe, A. Zimmerman. Expressive Alignment Language and Implementation. Project Deliverable D2.2.10, Knowledge Web NoE (EU-IST-2004-507482), 2007.
12. W. Faber, N. Leone, G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. *9th European Conference on Artificial Intelligence (JELIA 2004)*. Lisbon Portugal, 2004.
13. M. Gelfond, V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
14. C. Ghidini, F. Giunchiglia. Local model semantics, or contextual reasoning = locality + compatibility. *Artificial Intelligence*, 127(2):221–259, 2001.
15. P. Hayes. RDF Semantics. Technical Report, W3C, February 2004. W3C Recommendation.
16. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. W3C Member Submission.
17. C. Kiefer, A. Bernstein, H. J. Lee, M. Klein, M. Stocker. Semantic Process Retrieval with iSPARQL. *4th European Semantic Web Conference (ESWC '07)*. Innsbruck, Austria, 2007.
18. M. Kifer, G. Lausen, J. Wu. Logical Foundations of Object-oriented and Frame-based Languages. *Journal of the ACM*, 42(4):741–843, 1995.
19. A. Malhotra, J. Melton, N. W. (eds.). XQuery 1.0 and XPath 2.0 Functions and Operators, Jan. 2007. W3C Recommendation.
20. S. Muñoz, J. Pérez, C. Gutierrez. Minimal Deductive Systems for RDF. *4th European Semantic Web Conference (ESWC'07)*, Innsbruck, Austria, 2007.
21. J. Pérez, M. Arenas, C. Gutierrez. Semantics and Complexity of SPARQL. In *International Semantic Web Conference (ISWC 2006)*, pp. 30–43, 2006.
22. A. Polleres. From SPARQL to Rules (and back). *16th World Wide Web Conference (WWW2007)*, Banff, Canada, May 2007.
23. A. Polleres, R. Schindlauer. dlvhex-sparql: A SPARQL-compliant Query Engine based on dlvhex. *2nd Int. Workshop on Applications of Logic Programming to the Web, Semantic Web and Web Services (ALPSWS2007)*, Porto, Portugal, 2007.
24. E. Prud'hommeaux, A. Seaborne (eds.). SPARQL Query Language for RDF, June 2007. W3C Candidate Recommendation.
25. F. Scharffe, J. de Bruijn. A Language to specify Mappings between Ontologies. In *First Int. Conf. on Signal-Image Technology and Internet-Based Systems (IEEE SITIS05)*, 2005.
26. S. Schenk, S. Staab. Networked rdf graphs. Tech. Report, Univ. Koblenz, 2007. `http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf`.
27. R. Schindlauer. *Answer-Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Dec. 2006.
28. J. Ullman. *Principles of Database & Knowledge Base Systems*. Comp. Science Press, 1989.