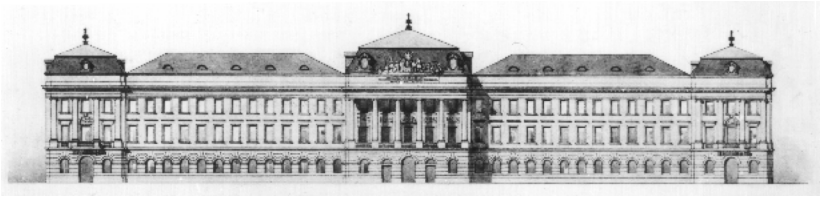
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

On the relation between SPARQL1.1 and Answer Set Programming

DBAI-TR-2013-84

Axel Polleres **Johannes P. Wallner**

DBAI TECHNICAL REPORT

2013

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2013-84, 2013

On the relation between SPARQL1.1 and Answer Set Programming

Axel Polleres [1] **Johannes P. Wallner** [2]

Abstract. In the context of the emerging Semantic Web and the quest for a common logical framework underpinning its architecture, the relation of rule-based languages such as Answer Set Programming (ASP) and ontology languages such as OWL has attracted a lot of attention in the literature over the past years. With its roots in Deductive Databases and Datalog though, ASP shares much more commonality with another Semantic Web standard, namely the query language SPARQL. In this paper, we take the recent approval of the SPARQL1.1 standard by the World Wide Web consortium (W3C) as an opportunity to introduce this standard to the Logic Programming community by providing a translation of SPARQL1.1 into ASP. In this translation, we explain and highlight peculiarities of the new W3C standard. Along the way, we survey existing literature on foundations of SPARQL and SPARQL1.1, and also combinations of SPARQL with ontology and rules languages. Thereby, apart from providing means to implement and support SPARQL natively within Logic Programming engines and particularly ASP engines, we hope to pave the way for further research on a common logical framework for Semantic Web languages, including query languages, from an ASP point of view.

[1]Vienna University of Economics and Business (WU Wien), Welthandelsplatz 1, 1020 Vienna, Austria E-mail: axel.polleres@wu.ac.at

[2]Institute for Information Systems 184/2, Technische Universität Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: wallner@dbai.tuwien.ac.at

A journal version of this article has been published in JANCL. Please cite as:
A. Polleres and J.P. Wallner. On the relation between SPARQL1.1 and Answer Set Programming. Journal of Applied Non-Classical Logics (JANCL), 23(1-2):159-212, 2013. Special issue on Equilibrium Logic and Answer Set Programming.

Contents

1 Introduction

The Semantic Web is, in principle, a family of standards to enable a Web of Data, with the final goal of enabling nothing less than the vision of the Web as "one huge database" (Berners-Lee, 1999). Whereas the Semantic Web architecture is often depicted as a stack[1], we dispense with this metaphor herein and just summarize the basic components needed for such a "Web database" and its related W3C standards:

Data model The data model of the Semantic Web is a simple, schema-less model made up by the *Resources Description Framework (RDF)* (Manola & Miller, 2004), where all data is expressed in the form of subject-predicate-object triples that consist of resources identifiable by URIs. RDF was released as a standard first in 1999, with a formal model-theoretic semantics (Hayes, 2004) being added in 2004; the next upcoming version of RDF is currently being worked on in the RDF1.1 working group.[2]

Schema languages In order to add schema-information to RDF, the W3C additionally defined schema languages such as *RDF Schema (RDFS)* (Brickley & Guha, 2004), and the *Web Ontology Language (OWL)* (Smith, Welty, & McGuinness, 2004), both first released in 2004. OWL was recently subsumed by the OWL2 (W3C OWL 2 Working Group, 2012) specification. RDF Schema is a fairly simple language, mainly allowing to express class membership of resources, simple class hierarchies, and role hierarchies for resources used as predicates in RDF triples. On the contrary, OWL is a more sophisticated ontology language, that allows to express more complex relations between classes and roles, with its semantic roots in Description Logics (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003).

Rule languages Despite the fact that OWL and RDF Schema can express various implicit information in RDF Data, the lack of (built-in) functions, default negation (aka negation as failure, i.e., adding non-monotonicity) and other features common in rule-based languages significantly limited the ability to express implicit information in RDF. Acknowledging the existence of various rule-based formalisms with different semantics to fill this gap, each useful in its own right, instead of standardizing a single rule-based formalism, the W3C standardized a so-called *Rule-Interchange Format (RIF)* (Kifer & Boley, 2012; Boley, Kifer, Pătrânjan, & Polleres, 2007) that comprises a number of extensible dialects to encode and exchange rule-based knowledge.

Query language Finally, in order to access data described in RDF and described in terms of rules and ontologies, a standard query language was needed, a gap finally filled in 2008 with the standardization of SPARQL (Prud′hommeaux & Seaborne, 2008), the "Simple Protocol and RDF Query Language". The new version of the SPARQL query language, SPARQL1.1 (Harris & Seaborne, 2013), which adds a lot of new features, just has reached W3C recommendation status.

[1] various incarnations of this "stack" – which has significantly changed during the evolution of Semantic Web standards – are for instance collected in (Polleres, 2013).

[2] Within this paper, we will restrict ourselves to the current 2004 version of the standard.

Lately, also the Linked Data principles (Berners-Lee, 2006; Heath & Bizer, 2011) – a set of best practices to publish RDF data online in a coherent manner using URIs as cross-references to indeed enable a *Web* of data – are often counted into the basic components of the Semantic Web. However, as far the present paper is concerned we want to restrict the discussion herein to the above four components, and particularly to the query language.

Concerning the interplay of these components within the common Semantic Web architecture, a majority of the recent discussions have revolved around the combinations of rules languages and schema languages: the RIF standard comprises a standard interfacing mechanism between RIF and OWL (de Bruijn, 2010), where reasoning tasks such as entailment become undecidable in general. In this context, a lot of discussion and academic works have been devoted over the last years to the theoretical combinations of Description Logics based schema languages such as OWL and rule-based languages (expressible in RIF), cf. (Levy & Rousset, 1998; Grosof, Horrocks, Volz, & Decker, 2003; Motik, Sattler, & Studer, 2005; Krötzsch, Rudolph, & Hitzler, 2007a; Rosati, 2005a, 2005b, 2006b; Motik & Rosati, 2007; Lukasiewicz, 2010; Eiter, Lukasiewicz, Schindlauer, & Tompits, 2004; Motik & Rosati, 2007; de Bruijn, Eiter, Polleres, & Tompits, 2007; de Bruijn, Pearce, Polleres, & Valverde, 2010).[3] While some of these approaches focus on decidable combinations due to narrowing down the interface between these languages, others are concerned with embedding them into a common logical formalism. Particularly, (de Bruijn et al., 2010) suggests a quantified version of Equilibrium Logic, which is also the underlying logic of Answer Set Programming for this common logical formalism.

Meanwhile also that interplay of SPARQL with the other components of the Semantic Web architecture are gaining more attention: the new SPARQL1.1 specification defines a standard mechanism to respect entailments coming from RDF Schema, OWL, or RIF in SPARQL query results (Glimm et al., 2013; Kollia, Glimm, & Horrocks, 2011). As for academia, several foundational works have contributed to clarifying relations between SPARQL and its neighboring standards. The original formal semantics of SPARQL is very much inspired by academic results, such as by the seminal papers of Pérez et al. (Pérez, Arenas, & Gutierrez, 2006, 2009). Based on this work first results on the relation of SPARQL and Datalog (with default negation) were published in (Polleres, 2007) and later extended in (Angles & Gutierrez, 2008): Angles and Gutierrez showed that SPARQL has exactly the expressive power of non-recursive safe Datalog with default negation, and thus is also embeddable in ASP. However, all these results are so far based on the "academic" semantics of SPARQL, which is only to some extent compatible with the official semantics as defined by the W3C: that is, the work of (Pérez et al., 2009) deals with a set semantics, while the official one is based on a bag semantics, where duplicates are allowed. Likewise, the results with respect to expressivity in (Angles & Gutierrez, 2008) and translations from SPARQL to Datalog (Polleres, 2007) are based on (Pérez et al., 2006). The differences between set semantics and bag semantics were partially discussed in (Polleres & Schindlauer, 2007; Arenas, Gutierrez, & Pérez, 2009).

Subsequently several extensions, which are incorporated partially in SPARQL1.1, were investigated in the academic literature, such as top-level filters in optional patterns by (Angles & Gutier-

[3]For recent surveys, we refer the interested reader additionally to (Eiter, Ianni, Polleres, Schindlauer, & Tompits, 2006; Eiter, Ianni, Krennwallner, & Polleres, 2008; Rosati, 2006a; Krisnadhi, Maier, & Hitzler, 2011).

rez, 2008), subqueries (Angles & Gutierrez, 2011), assignments (Polleres, Scharffe, & Schindlauer, 2007), and property paths (Alkhateeb, Baget, & Euzenat, 2009; Pérez, Arenas, & Gutierrez, 2008; Arenas, Conca, & Pérez, 2012; Losemann & Martens, 2012). Note that the semantics of these features in the official SPARQL1.1 specification again slightly differs from the discussions in these papers. Also, an embedding of the new features of SPARQL1.1 into Datalog and ASP is not yet covered in these works.

Here is where the present paper steps in: this work may be viewed as an extended version of (Polleres, 2007), where we aim to reconcile and extend the results in a twofold manner:

- Firstly, we give a full formal account of the SPARQL semantics compatible with the SPARQL W3C specification based on the existing academic literature, where (a) we highlight non-obvious peculiarities in the specification such as subtleties in MINUS patterns, and (b) we consider the new features of SPARQL1.1, namely solution modifiers, property paths, subqueries, and value creation through (BIND).

- We provide a systematic translation from most of SPARQL1.1 to Answer Set Programming, where it is our goal to stick to the best of our knowledge to the official semantics of the specification.[4]

We believe this work is valuable in providing both (i) means to implement and support SPARQL1.1 natively within Logic Programming engines and particularly ASP engines and (ii) a basis for further research on a common logical framework for Semantic Web languages, including query languages, grounded in ASP.

This paper is structured as follows: In Section 2 we first provide an overview of RDF and SPARQL1.1, discussing graph patterns in Section 2.2, datasets in Section 2.1, solution modifiers in Section 2.3, followed by some issues in the language (Section 2.4). We then define SPARQL's formal semantics in Section 2.5. After introducing various fragments of ASP in Section 3, we proceed with the translations of SPARQL to ASP in Section 4, starting with the core translation for SPARQL features in Section 4.1, followed by the new SPARQL1.1 translation in Section 4.3 and finally the translation of the BIND patterns – which we keep separate since, as we will see, these require built-ins that cannot be modeled in "pure" ASP – in Section 4.4. Finally we present some variants of the translation for other query forms of SPARQL in Section 4.5, discuss interesting directions and related work in Section 5, and conclude in Section 6.

2 RDF and SPARQL

We assume basic familiarity with the Resource Description Framework (RDF) (Manola & Miller, 2004; Hayes, 2004) and the Turtle (Beckett & Berners-Lee, 2008) syntax for RDF and will keep formal definitions to the necessary minimum for the moment. RDF is essentially a data format that consists simply of subject-predicate-object triples (also called statements). Turtle is a concise

[4]Minor differences, including coverage of FILTER functions or taking particular choices in cases where the specification leaves different routes for implementations will be pointed at in the text.

notation for such RDF triples where triples are simply separated by '.', predicate-object groups for the same subject can be grouped using ';', and blank nodes (i.e. existential variables scoped over a graph), are written as `_:x`.

In examples we will subsequently refer to the two RDF graphs in Figure 1 which give some information about Bob and $Alice$. Such information about persons is common in so-called FOAF (Brickley & Miller, 2007) profiles which allow to describe personal data and social networks on the Web. Similarities with existing examples in (Harris & Seaborne, 2013) are on purpose. The two RDF graphs are given in Turtle notation and we assume they are accessible via the IRIs (RFC-3987, 2005) `ex.org/bob` and `alice.org`.[5]

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

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

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

```
# Graph: alice.org

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

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

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

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

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

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

Figure 1: Two RDF graphs in Turtle notation and a simple SPARQL query.

We assume the pairwise disjoint, infinite sets I, B, L and Var, which denote IRIs, Blank nodes, RDF literals, and variables respectively. In this paper an *RDF Graph* is then a finite set of triples from $(I \cup B \cup L) \times (I) \times (I \cup B \cup L)$,[6] dereferenceable by an IRI. A SPARQL *query* is a tuple $Q = (DS, P, SM)$, where DS is a dataset, P is a graph pattern, and SM is a solution modifier. In the following sections we will formally introduce the language of SPARQL interleaved with intuitive examples for many of the prominent features and then proceed to the formal definition of the semantics of SPARQL queries. We will formally define datasets in Section 2.1 and graph patterns in Section 2.2, where we first describe graph patterns composed of patterns available in SPARQL1.0 and then extend these with the enriched features of the new SPARQL1.1 version. Solution modifiers are then formally introduced in Section 2.3. For further syntactical details on SPARQL queries we refer the interested reader to (Harris & Seaborne, 2013). Subsequently

[5]For reasons of legibility and conciseness, we normally omit the leading 'http://' or other schema identifiers in IRIs in this paper, IRIs and literals can be disambiguated by enclosing quotes for the latter.

[6]Following SPARQL, we are slightly more general than the original RDF specification in that we allow literals in subject positions. Note further that in this paper we only consider plain literals, i.e., literals without language tags or datatypes.

we remark necessary preprocessing steps for the formalism used in this work in Section 2.4 and afterwards we define and extend the formal semantics from (Arenas et al., 2009) in Section 2.5.

2.1 Datasets

A dataset in a SPARQL query is used to define which graphs should be used for matching. The dataset of a SPARQL query is defined by a default graph G plus a set of named graphs, i.e. pairs of IRIs (denoting "names") and their corresponding graphs.

Definition 2.1 (Dataset, default graph, named graph) *Let $G, G_1, \ldots, G_k$ be RDF graphs and $g_1, \ldots, g_k$ be IRIs, then $DS = (G, \{(g_1, G_1), \ldots (g_k, G_k)\})$ is a* dataset. *We call G the* default graph *and $(g_1, G_1), \ldots, (g_k, G_k)$* named graphs.

A basic operation on a set of graphs is the graph merge, denoted by '$\uplus$', as defined in (Hayes, 2004, Section 0.3), which is the union of the graphs, i.e. the union of the triples, if the sets of blank nodes from each graph is disjoint. If the sets are not disjoint, then the graphs are first replaced by equivalent graphs with disjoint sets of blank nodes.

Without loss of generality (there are other ways to define the dataset such as in a SPARQL protocol query), we assume that the default graph G in a dataset is given as the merge of the graphs denoted by the IRIs given in a set of FROM clauses and the named graphs given by a set of FROM NAMED clauses.

For instance, the query from Figure 1 refers to the dataset which consists of the default graph obtained from merging `alice.org` $\uplus$ `ex.org/bob` plus an empty set of named graphs.

The relation between names and graphs in SPARQL datasets is defined solely in terms of that the IRI defines a resource which is represented by the respective graph. In this paper, we assume that the IRIs represent indeed network-accessible resources where the respective RDF graphs can be retrieved from (using, e.g., the HTTP protocol). Particularly, this treatment is not to be confused with so-called named graphs in the sense of (Carroll, Bizer, Hayes, & Stickler, 2005). We thus identify each IRI with the RDF graph available at this IRI and each set G of IRIs with the graph merge over the graphs accessible at the respective IRIs in this set G. This allows us to identify the dataset by a pair of sets of IRIs $DS = (G, G_{named})$ with $G = \{d_1, \ldots, d_n\}$ and $G_{named} = \{g_1, \ldots, g_k\}$ denoting the (merged) default graph and the set of named graphs, respectively. Hence, the dataset referred to in the query of Figure 1 may be written as $DS_1 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$ whereas the following set of clauses

```
FROM <ex.org/bob>
FROM NAMED <alice.org>
```

defines the dataset $DS_2 = (\{\texttt{ex.org/bob}\}, \{\texttt{alice.org}\})$.

2.2 Graph Patterns

Graph patterns are the fundamental building blocks from which SPARQL queries are composed of. These patterns are based on the idea of graph pattern matching. Basic graph patterns directly

match triples in the given RDF Graph. More complex patterns support other operations on the solution set, such as relational join, union and optional values. We follow the recursive definition of graph patterns P from (Arenas et al., 2009) and will first introduce graph patterns available in SPARQL1.0 and later extend these to SPARQL1.1.

Definition 2.2 (SPARQL**1.0 graph pattern)** *A* SPARQL*1.0 graph pattern P is recursively defined as follows:*

- *a tuple (s, p, o) is a graph pattern (also called* triple pattern*) where $s, o \in I \cup L \cup Var$ and $p \in I \cup Var$.*[7]
- *if P and P' are graph patterns then $(P$ AND $P')$, $(P$ UNION $P')$ and $(P$ OPT $P')$ are graph patterns.*[8]
- *if P is a graph pattern and $i \in I \cup Var$ then $(\mathsf{GRAPH}\, i\ P)$ is a graph pattern.*
- *if P is a graph pattern and R is a filter expression then $(P$ FILTER $R)$ is a graph pattern.*

For any pattern P, we denote by $vars(P)$ the set of all variables occurring in triple patterns within P, that is variables only occurring in filter expressions do not count among $vars(P)$. Analogously, for any filter expression R, we denote by $vars(R)$ the set of all variables occurring in R.

The GRAPH pattern is used to match patterns against named graphs. FILTER expressions restrict the solutions to those satisfying the condition. As *atomic filter expression*, SPARQL allows the unary predicates BOUND, isBLANK, isIRI, isLITERAL, as well as binary equality ('='), and a and a binary comparison operator ('<'). The specification includes more operators. We refer the reader to (Harris & Seaborne, 2013, Section 17.3) for more details. Without loss of generality, we assume only variables and constants to appear in such comparisons; as for values within comparisons that arise from complex built-in expressions by e.g. string functions and arithmetic operators, these can be emulated by assignments (BIND), see the remarks on Example 2.5 below. *Complex filter expressions* can be built using logical connectives and auxiliary parentheses.

Definition 2.3 (Filter expression) *Let $x, y \in I \cup L \cup Var$. Then a filter expression R is recursively defined as follows:*

- *$\mathsf{BOUND}(x)$, $\mathsf{isBLANK}(x)$, $\mathsf{isIRI}(x)$, $\mathsf{isLITERAL}(x)$, $(x = y)$, $(x < y)$ are (atomic) filter expressions.*
- *if R is a filter expression, then $(\neg R)$ is a (complex) filter expression.*
- *if R, R' are filter expressions, then $(R \circ R')$ is a (complex) filter expression for $\circ \in \{\wedge, \vee\}$.*

[7] We do not consider blank nodes in patterns as these can be semantically equivalently replaced by variables in graph patterns (de Bruijn, Franconi, & Tessaris, 2005).

[8] Note that AND is not a designated keyword in SPARQL's syntax, but we use it here for reasons of readability and in order to keep with the operator style definition of (Arenas et al., 2009).

Moreover, note that the semantics defines errors for FILTERs to avoid mistyped comparisons, or evaluation of built-in functions over unbound values. These filters rely on a three-valued logic $(\top, \bot, \varepsilon)$, where errors (ε) propagate over the whole FILTER expression, also over negation and conjunction (but not over disjunction, cf. the truth tables in (Harris & Seaborne, 2013, Section 17.2)). That is, for instance the negation of an error yields an error in its turn, as shown by the following example.

Example 2.1 *Assuming the dataset does not contain triples for the* `foaf:nick` *property, the example query*

```
SELECT ?X
WHERE { {?X a foaf:Person .
           OPT { ?X foaf:nick ?Y . } }
         FILTER ( ¬(isLITERAL (?Y)) ) }
```

would discard any solution for ?X*, since the unbound value for* ?Y *causes an error in the* isLITERAL *expression and thus the whole* FILTER *expression returns an error.*

The translation of patterns as written in WHERE clauses in normal SPARQL queries to the format we use here is straightforward, with the exception that FILTERs may occur anywhere within a group of graph patterns, not only in the end. In the more algebraic notation we use here, all the non-FILTER parts of such a group would be conjoined by AND and all FILTERs conjoined with $\wedge$ in the end, such that without loss of generality we can assume that each group contains at most one FILTER in the end. In Section 2.4 we discuss further preprocessing steps required to capture special cases of the SPARQL specification.

The following example illustrates this transformation from an actual WHERE clause in SPARQL's surface syntax to the format used herein for patterns:

Example 2.2 *Let us consider the following two queries over dataset* $DS = (\{\texttt{ex.org/bob}\}, \emptyset)$.

```
SELECT ?N WHERE { ?G foaf:name ?N . FILTER( !( ?N = ?FN ) )
                  ?G foaf:knows ?F . FILTER( ?FN="Alice" )
                  ?F foaf:nick ?FN }
```

This corresponds to the query $Q = (DS, P, SM)$ *such that*

$$P = (((?G, \texttt{foaf:name}, ?N) \textit{ AND } (?G, \texttt{foaf:knows}, ?F) \textit{ AND } (?F, \texttt{foaf:nick}, ?FN)) \textit{ FILTER } (\neg(?N = ?FN) \wedge (?FN = \text{"}\texttt{Alice}\text{"}))).$$

Likewise,

```
SELECT ?N  WHERE { { ?G foaf:name ?N . FILTER( !( ?N = ?FN ) )
                     ?G foaf:knows ?F . FILTER( ?FN="Alice" ) }
                   ?F foaf:nick ?FN }
```

corresponds to $Q' = (DS, P', SM)$ *such that*

$$P' = ((((?G, \texttt{foaf:name}, ?N)\ \mathsf{AND}\ (?G, \texttt{foaf:knows}, ?F))\\ \mathsf{FILTER}\ (\neg(?N = ?FN) \wedge (?FN = \text{"}\texttt{Alice}\text{"})))\\ \mathsf{AND}\ (?F, \texttt{foaf:nick}, ?FN))$$

That is, filter expressions are always evaluated in the scope of the "{", "}" delimited group they occur in.

Newly available graph patterns in SPARQL1.1 are subqueries, property paths, two forms of negation (MINUS, ! EXISTS[9]) and assignment (BIND). Subqueries enhance SPARQL by adding SELECT subqueries at any position of the query where graph patterns are allowed, including solution modifiers within such subqueries. Property paths augment the query language by regular expressions. Negation can be used to exclude solutions; assignments are useful for introducing new values computed by an expression.

A property path PP incorporates regular expressions on IRIs applying the widely known operators for negation '!', inverse '^', sequence '/' and alternatives '|'. For variable length SPARQL supports the operators for the zero-or-more '*', the one-or-more '+' and the zero-or-one '?' path.

Definition 2.4 (Property path) *A property path PP is recursively defined as follows:*

- $p \in I$ *is a property path.*
- *if PP_1 and PP_2 are property paths, then PP_1/PP_2 and $PP_1|PP_2$ are property paths, also called sequential and alternative property paths, respectively.*
- *if PP is a property path, then $PP\circ$ for $\circ \in \{?, *, +\}$ is a property path.*
- *if PP is a property path, then $\hat{}PP$ is a property path.*
- *if $p_i \in I$ for $1 \leq i \leq n + m$, then $!PP$ is a property path where $PP = (p_1|\cdots|p_n|\hat{}p_{n+1}|\cdots|\hat{}p_m)$*

That is, property path patterns generalize the usual triple patterns. For readability we will identify the negated property path with a negated pair of two sets of IRIS, i.e. $!(PP) = !(N, N')$, where the first, N, contains only normal IRIs and the latter, N', the inverted IRIs. Now SPARQL1.1 graph patterns are extended in the following definition.

Definition 2.5 (SPARQL**1.1 graph pattern)** *A* SPARQL*1.1 graph pattern P is recursively defined as follows:*

- *a* SPARQL*1.0 graph pattern P is a graph pattern.*

[9] which can be written similar to SQL as "NOT EXISTS")

- *if SM is a solution modifier, then a subquery (P, SM) is a graph pattern.*
- *a tuple (s, PP, o) is a graph pattern (also called* property path pattern*) where $s, o \in I \cup L \cup Var$ and PP a property path.*
- *if P and P' are graph patterns, then $(P$ MINUS $P')$ is a graph pattern*
- *if P is a graph pattern, $x \in Var$ and $expr$ is an assignment expression, then $(P$ BIND $expr$ AS $x)$ is a graph pattern.*
- *if $\vec{V} \in Var^n$ is a list of variables and D a finite set of tuples $(I \cup B \cup L)^n$ with arity n, then $(\mathsf{VALUES}\ \vec{V}\ D)$ is a graph pattern.*

Note that we define solution modifiers in Section 2.3. In the following we will exemplify the new features, starting with property paths.

Example 2.3 *Consider one wants to find all persons Alice knows and recursively the persons they in turn know and finally retrieve their names. This query can easily be specified using property paths, over dataset $DS = (\{ex.org/bob, \emptyset\})$.*

```
SELECT ?N WHERE { ?F foaf:name ?N .
                  ?A foaf:knows+ ?F .
                  ?A foaf:name "Alice"}
```

Composing the property path additionally with a sequential path gives the opportunity to simplify this further as:

```
SELECT ?N WHERE { ?A foaf:knows+/foaf:name ?N .
                   ?A foaf:name "Alice"}
```

The negation support in SPARQL1.1 has been enhanced by adding the MINUS graph pattern and a new atomic filter expression, namely EXISTS(P) for any graph pattern P. The latter can be negated as normal by '$\neg$' in a complex filter expression.

Example 2.4 *Extracting all persons, who know Bob from the example graph can be done in several ways. One possibility is to query for all persons, where there exists a connection to Bob as follows, over dataset $DS = (\{alice.org, \emptyset\})$.*

```
SELECT ?N WHERE { ?P foaf:name ?N .
                  FILTER ( EXISTS (?P foaf:knows ?F .
                                   ?F foaf:name "Bob"))}
```

For assignments SPARQL1.1 features two methods, first a straightforward way of listing inline data for variables via the VALUES keyword and on the other hand the BIND pattern, which assigns the value of an assignment expression to a variable x. The specification supports many different types of expressions (Harris & Seaborne, 2013, Section 17), such as string functions, arithmetic operators, etc. For the purpose of this paper we leave the exact nature of this assignment expression open and will later on rely on an oracle function, which can handle (i.e., evaluate) these expressions.

Assignments can not only occur natively in the form of a BIND graph pattern, but also inside projected variables in SELECT clauses. These can be equivalently rewritten to BIND graph patterns.

Example 2.5 *Consider we want to query for all persons in the graph and output their name and nickname in one field, that is concatenating both results, over dataset* $DS = (\{alice.org, \emptyset\})$.

```
SELECT (concat(?M," a.k.a. ",?N) AS ?X) WHERE { ?P foaf:nick ?N .
                                                ?P foaf:name ?M }
```

This query outputs e.g. "Bob a.k.a. Bobby", with the delimiter set in the SELECT clause. We can rewrite this query to one using BIND, with a fresh variable.

```
SELECT ?X WHERE { ?P foaf:nick ?N .
                  ?P foaf:name ?M .
                  BIND (concat(?M," a.k.a. ",?N) AS ?X) }
```

We will thus only discuss BIND in the remainder for assignments, but not assignments within SELECT clauses. Note that a built-in expression occurring within a FILTER, e.g.

```
FILTER ( concat(?M," a.k.a. ",?N) != " a.k.a. ")
```

can be similarly "emulated" by inserting a BIND (and introducing a new variable) before the FILTER, i.e.

```
BIND (concat(?M," a.k.a. ",?N) AS ?New) FILTER (?New != " a.k.a. ")
```

2.3 Solution Modifiers

Query results in SPARQL are given by partial, i.e. possibly incomplete, substitutions of variables in $vars(P)$ by RDF terms occurring in the graphs of DS. In traditional relational query languages such incompleteness is usually expressed using null values. Using such null values we will write solutions as tuples where the order of columns is determined by *lexicographically ordering* the variables in P. Given a set of variables V, let $\overline{V}$ denote the tuple obtained from lexicographically ordering V.

The query from Figure 1 has three solution tuples. We write substitutions – which are partial mappings from variables to RDF terms – in square brackets, so these tuples correspond to the substitutions $\theta_1 = [?X \to$ "Alice", $?Y \to$ alice.org#me$]$, $\theta_2 = [?X \to$ "Bob", $?Y \to$ _:a$]$,

and $\theta_3 = [?X \to$ "Bob", $?Y \to$ _ : c], respectively. A more formal definition will be given in Section 2.5. The short notation ("Bob", _:a), ("Alice", alice.org#me), ("Bob", _:c) which we sometimes use here is obtained from applying these substitutions to the result form $\overline{V} = (?X, ?Y)$.

In order to only project solutions to particular variables, sort solutions, or only request a certain number of results, SPARQL provides so-called solution modifiers indicated by the SELECT clause, as well as the keywords DISTINCT, ORDER BY, LIMIT, and OFFSET. In this paper we will mainly deal with SELECT queries. Other result forms are discussed in Section 4.5.

In its simplest form, the solution modifier SM consists only of a a set $V \subseteq vars(P)$ of variables marking a projection (SELECT clause). The optional DISTINCT keyword indicates that duplicates shall be eliminated during this projection. Another optional part of the solution modifier is an ORDER BY clause, which puts the solution tuples to a query in an order.[10]

For the purposes of this paper, we assume the optional ORDER BY clause to refer to a list of variables from V, which determines by which bindings the solutions should be ordered, before limit or offset are applied. Ordering solutions using complex expressions can be "emulated" via a BIND assignment, similar as shown in Example 2.5 for assignment expressions in the SELECT clause. In order to guarantee deterministic results of queries wherever order plays a role (which is not guaranteed in SPARQL in general), we assume a *default order* that orders solutions lexicographically according to the bindings of the variables in $\overline{vars(P)}$. That is, for instance for the query from Figure 1, the default order will order solutions by the bindings for $?X$ first, if present, and then by bindings to $?Y$, which yields $\theta_1, \theta_2, \theta_3$ in that order as above. If we added an ORDER BY $?Y$ clause, the solution order would change to $\theta_2, \theta_3, \theta_1$, as blank nodes are ordered before IRIs in SPARQL.[11] Then the default order is changed to first ordering by bindings of ?Y and then ?X.

The solution modifiers LIMIT l and OFFSET o are used to exclude solutions by applying the so-called "slice", i.e. omitting the first o solutions and then outputting only the next l solutions after ordering has been applied. Formally we collect all the modifiers in a quintuple SM with the following definition.

Definition 2.6 (Solution modifier) *Let $\vec{V}$ be a vector of variables, $S \subseteq Var$ a set of variables, $dst \in \{true, false\}$, l and o positive integers. Then $SM = (\vec{V}, S, dst, l, o)$ is a solution modifier.*

In Section 2.5 we will formally define the semantics of solution modifiers. Just note that the solutions will be ordered accordingly to $\vec{V}$, the set S will act as the set of variables to project and dst is used for discarding duplicates. The following exemplifies uses for solution modifiers.

Example 2.6 *For instance, let us consider the following two queries over dataset $DS = (\{$ex.org/bob, alice.org$\}, \emptyset)$ asking for the second name or nickname (ordering lexicographically.)*

[10] Within the present paper we restrict ourselves to ascending order in ORDER BY clauses, but note that our approach can be easily generalized to also allow descending order, cf. keyword DESC() in (Harris & Seaborne, 2013, Section 15.1).

[11] Note that SPARQL does actually not impose an order between blank nodes _:c and _:a, but we can do so without changing the semantics.

```
SELECT ?N WHERE
{ { ?P foaf:name ?M . } UNION { ?P foaf:nick ?N . } }
ORDER BY ?N OFFSET 2 LIMIT 1
```

which will return the single substitution $[?N \to$ "`Alice`"$]$*, whereas the modified query*

```
SELECT DISTINCT ?N
WHERE { { ?P foaf:name ?M . } UNION { ?P foaf:nick ?N . } }
ORDER BY ?N OFFSET 2 LIMIT 1
```

will return the substitution $[?N \to$ "`Bob`"$]$.

That is, the DISTINCT modifier is applied before computing the "slice" of solutions determined by the ORDER BY, OFFSET, and LIMIT modifiers. This corresponds to the solution modifier $SM = ((?N, ?P), \{?N\}, true, 1, 2)$.

2.4 Query Preprocessing

We already mentioned the translation of SPARQL queries into the formalism of this paper and that complex built-in expressions in SELECT, FILTER and ORDER BY clauses need to be rewritten into graph patterns involving BIND and an auxiliary variable for this purpose. Additionally some more remarks are in place concerning FILTER expressions and a required rewriting. Graph patterns of the form $(P$ FILTER $R)$ may have *safe filter expressions*, i.e. that all variables used in a filter expression R also appear in the corresponding pattern P. This corresponds roughly to the notion of safety in ASP (see Section 3 below), where built-in predicates (which obviously correspond to filter predicates) do not suffice to safe unbound variables. We note that – as shown by Angles and Gutierrez (Angles & Gutierrez, 2008) – such unsafe variables in FILTERs do not affect the expressivity of SPARQL, i.e., each query containing unsafe FILTER variables can be equivalently rewritten to a query without such unsafe variables; notably however, (Angles & Gutierrez, 2008) do not consider the three-valued semantics of FILTER evaluations as per the official SPARQL specification, cf. (Harris & Seaborne, 2013, Section 17.2); we take a slightly different approach herein: since unsafe FILTER variables in SPARQL are treated as unbound, in our setting we can avoid unsafe FILTERs by just replacing unsafe variables in FILTER expressions with the constant null in a preprocessing step and, as we will see, will treat the null constant appropriately in our translation of filter expressions to ASP, see Section 4.1.

Example 2.7 *As an example for an unsafe variable in a FILTER, the following query*[12]

```
SELECT * WHERE  ?X foaf:mbox ?M . FILTER (?Name = "Alice")
```

simply amounts to

```
SELECT * WHERE  ?X foaf:mbox ?M . FILTER (null = "Alice")
```

[12] By the asterisk in a query "SELECT * WHERE P" one implicitly sets the result form to $vars(P)$.

Since we define – in conformance with the treatment of unbound variables in the official SPARQL *specification – comparisons with* null *to result in an error, the evaluation of such a FILTER expression will always fail, and thus the query returns an empty result.*

FILTER expressions within OPT patterns, as well as in patterns inside EXISTS expressions, cannot be quite treated this way, since top-level FILTERs within OPTs are allowed to refer to bindings from outside the OPT pattern. This is also discussed in (Angles & Gutierrez, 2008). This behavior is exemplified by the following query.

Example 2.8 *The following query is a variant of Example 2.7:*

```
SELECT * WHERE
  { ?X a foaf:Person . ?X foaf:name ?Name .
    OPT { ?X foaf:mbox ?M . FILTER (?Name = "Alice") } }
```

What this query intuitively expresses is the following: "Output all persons and their names, and the mbox only for persons named Alice".

Here, intuitively (and also according to the semantics defined in the SPARQL *specification) the FILTER evaluation shall not fail by default but the idea behind this query is to out put the mailbox only for persons named Alice and suppress the mailboxes in the results for others. That is, on the test data from Figure 1, if we assume* $DS = (\{\texttt{alice.org}'\}, \emptyset)$*, where* `alice.org`$'$ *has two additional triples*

```
alice:me foaf:mbox "alice@ex.org"
_:c foaf:mbox "bob@ex.org"
```

the query would yield two solution tuples $(\text{"}\texttt{alice@ex.org}\text{"}, \text{"}\texttt{Alice}\text{"}, \texttt{alice.org\#me})$ *and* $(null, \text{"}\texttt{Bob}\text{"}, \texttt{alice.org\#me\_:c})$.

Similarly, if an EXISTS clause is present in the filter, then its pattern may contain variables bound directly outside the filter in the corresponding pattern. We will take special care of these cases, when defining the semantics later on.

Lastly we do not consider blank nodes in query patterns as these can be semantically equivalently replaced by variables in graph patterns (de Bruijn et al., 2005). However, blank nodes may appear in the dataset, and thus in query answers. Note that different graphs in the dataset might use the same identifiers for blank nodes. For simplicity, we assume that such ambiguities are already resolved, i.e. that different graphs in the dataset use different blank node identifiers. By this assumption, we may treat blank node identifiers largely like normal constants and we do not need to rename blank nodes identifiers in query results.

2.5 Formal Semantics of SPARQL

The definition of a formal semantics of SPARQL has been first tackled by Pérez et al. (Pérez et al., 2006), later been refined in (Arenas et al., 2009). We will base on these formalisations and extend them in order to capture version 1.1 of the SPARQL specification (Harris & Seaborne, 2013). Particularly, our definitions vary from (Pérez et al., 2006) in the way we define joining unbound variables (represented by the distinct constant null in our approach) and incorporate the treatment of top-level filters in OPT patterns,[13] solution modifiers, subqueries, property paths, negation and assignment features. The latter has already been discussed to some extent in (Polleres et al., 2007), while the MINUS pattern in (Harris & Seaborne, 2013, Section 18.5) differs from an earlier version discussed in (Polleres et al., 2007).

We prefer sticking with the notation and basing on Pérez et al.'s semantics (for its declarative nature) over the more algorithmically defined semantics in the official specification, wherever possible.

We denote by $T = I \cup B \cup L$ the set of RDF terms and by T_{null} the union of $T \cup \{\mathsf{null}\}$, where null is a dedicated constant denoting the unknown value not appearing in any of $I, B,$ or L, as commonly introduced when defining outer joins in relational algebra.

A *substitution* θ from Var to T_{null} is a partial function $\theta : Var \to T_{\mathsf{null}}$. We write substitutions in postfix notation: For a triple pattern $t = (s, p, o)$ we denote by $t\theta$ the triple $(s\theta, p\theta, o\theta)$ obtained by applying the substitution to all variables in t. The *domain* of θ, denoted by $dom(\theta)$, is the subset of Var where θ is defined. For a substitution θ and a set of variables $D \subseteq Var$ we define the substitution θ^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 θ_1 and θ_2 be substitutions, then $\theta_1 \cup \theta_2$ is the substitution obtained as follows:

$$x(\theta_1 \cup \theta_2) = \begin{cases} x\theta_1 \text{ if } x\theta_1 \text{ defined and } x\theta_2^{dom(\theta_1)} \in \{\mathsf{null}, x\theta_1\} \\ \text{else: } x\theta_2 \text{ if } x\theta_2 \text{ defined and } x\theta_1^{dom(\theta_2)} = \mathsf{null} \\ \text{else: undefined} \end{cases}$$

Thus, in the union of two substitutions defined values in one take precedence over null values of the other substitution. For instance, given the substitutions $\theta_1 = [?X \to "Alice", ?Y \to \_ : \mathsf{a}, ?Z \to \mathsf{null}]$ and $\theta_2 = [?U \to "Bob", ?X \to "Alice", ?Y \to \mathsf{null}]$ we get: $\theta_1 \cup \theta_2 = [?U \to "Bob", ?X \to "Alice", ?Y \to \_ : \mathsf{a}, ?Z \to \mathsf{null}]$

Now, similar to (Pérez et al., 2006), we define the notion of compatibility between substitutions:

Two substitutions θ_1 and θ_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 (Arenas et al., 2009) we define join, union, difference, and outer join between multisets of substitutions. We adhere to the multiset semantics, as defined by the W3C by using the ⊎ operator for multiset union which does not discard duplicates and the delimiters '{{' and '}}' for enumerations of multisets.

[13] This special treatment is discussed in (Angles & Gutierrez, 2008), but as opposed to their work, we define the special treatment of such filters in OPT directly in the semantics.

Definition 2.7 (SPARQL **Relational Algebra)** *Let Ω_1 and Ω_2 be multisets of substitutions over domains D_1 and D_2, respectively. Then we define the following basic operators*

$\Omega_1 \bowtie \Omega_2 = \{\!\{\theta_1 \cup \theta_2 \mid \theta_1 \in \Omega_1, \theta_2 \in \Omega_2, \textit{ are compatible}\}\!\}$
$\Omega_1 \Cup \Omega_2 = \{\!\{\theta \mid \exists\theta_1 \in \Omega_1 \textit{ with } \theta = \theta_1^{D_1 \cup D_2} \textit{ or } \exists\theta_2 \in \Omega_2 \textit{ with } \theta = \theta_2^{D_1 \cup D_2}\}\!\}$
$\Omega_1 - \Omega_2 = \{\!\{\theta_1 \in \Omega_1 \mid \textit{for all } \theta_2 \in \Omega_2,\ \theta_1 \textit{ and } \theta_2 \textit{ not compatible}\}\!\}$
$\Omega_1 \text{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \Cup (\Omega_1 - \Omega_2)$

The semantics of a graph pattern P over dataset $DS = (G, G_{named})$, can now be defined recursively by an evaluation function returning multisets of substitutions. Due to the variety of graph patterns we will split the definition into several logical parts as before in Section 2. We begin with SPARQL features (Prud'hommeaux & Seaborne, 2008).

Graph patterns and filter expressions are defined recursively based on the relational algebra.

Definition 2.8 (Evaluation of graph patterns, (Arenas et al., 2009, Definition 13.3)) *Let $t = (s, p, o)$ be a triple pattern, P, P_1, P_2 graph patterns, $DS = (G, G_{named})$ a dataset, then the evaluation $[[\cdot]]_{DS}$ is defined as follows:*

$[[t]]_{DS} = \{\theta \mid dom(\theta) = vars(t) \textit{ and } t\theta \in G\}$
$[[P_1 \textsf{ AND } P_2]]_{DS} = [[P_1]]_{DS} \bowtie [[P_2]]_{DS}$
$[[P_1 \textsf{ UNION } P_2]]_{DS} = [[P_1]]_{DS} \Cup [[P_2]]_{DS}$
$[[\textsf{GRAPH } i\ P]]_{DS} = [[P]]_{(i,\emptyset)}$, *for* $i \in G_{named}$
$[[\textsf{GRAPH } i\ P]]_{DS} = \emptyset$, *for* $i \in I - G_{named}$
$[[\textsf{GRAPH } v\ P]]_{DS} = \{\!\{\theta \cup [v \to g] \mid g \in G_{named} \textit{ and } \theta \in [[\ P[v \to g]\ ]]_{(g,\emptyset)}\}\!\}$, *for* $v \in Var$
$[[P \textsf{ FILTER } R]]_{DS} = \{\!\{\theta \in [[P]]_{DS} \mid R\theta = \top\}\!\}$

The semantics of FILTER expressions specifies whether a substitution θ satisfies a filter, does not satisfy it or results in an error. For this purpose we utilize a three valued logic, where '$\top$' stands for "true", '$\bot$' stands for "false" and 'ε' stands for errors, see (Harris & Seaborne, 2013, Section 17.3) and Example 2.1 for details.

Definition 2.9 (Evaluation of FILTER, extends (Arenas et al., 2009, Definition 13.4)) *Let R be a FILTER expression, $u, v \in Var \cup T_{\text{null}}$. The valuation of R on a substitution θ, written $R\theta$ takes one of the three values $\{\top, \bot, \varepsilon\}$ and is defined as follows.*
$R\theta = \top$, if:

(1) $R = \textsf{BOUND}(v)$ *with* $v \in dom(\theta) \ \wedge\ v\theta \neq \text{null}$;
(2) $R = \textsf{isBLANK}(v)$ *with* $v \in dom(\theta) \ \wedge\ v\theta \in B$;
(3) $R = \textsf{isIRI}(v)$ *with* $v \in dom(\theta) \ \wedge\ v\theta \in I$;
(4) $R = \textsf{isLITERAL}(v)$ *with* $v \in dom(\theta) \ \wedge\ v\theta \in L$;
(5) $R = (u = v)$ *with* $u, v \in dom(\theta) \cup T \wedge\ u\theta = v\theta \wedge u\theta \neq \text{null} \wedge v\theta \neq \text{null}$;
(6) $R = (u < v)$ *with* $u, v \in dom(\theta) \cup T \wedge\ u\theta < v\theta \wedge u\theta \neq \text{null} \wedge v\theta \neq \text{null}$;

(7) $R = (\neg R_1)$ *with* $(R_1, DS)\theta = \bot$*;*
(8) $R = (R1 \vee R2)$ *with* $R_1\theta = \top \ \vee \ R_2\theta = \top$*;*
(9) $R = (R1 \wedge R2)$ *with* $R_1\theta = \top \ \wedge \ R_2\theta = \top$*;*
(10) $R = \mathsf{EXISTS}(P)$ *with* $[[P\theta]]_{DS} \neq \emptyset$*.*

$R\theta = \varepsilon$*, if:*

(1) $R = \mathsf{isBLANK}(v)$,$R = \mathsf{isIRI}(v)$*, or* $R = \mathsf{isLITERAL}(v)$ *and*
$v \notin dom(\theta) \cup T \ \vee \ v\theta = \mathsf{null}$*;*
(2) $R = (u = v)$ *with* $u \notin dom(\theta) \cup T \ \vee \ u\theta = \mathsf{null} \vee v \notin dom(\theta) \cup T \ \vee \ v\theta = \mathsf{null}$*;*
(2) $R = (u < v)$ *with* $u \notin dom(\theta) \cup T \ \vee \ u\theta = \mathsf{null} \vee v \notin dom(\theta) \cup T \ \vee \ v\theta = \mathsf{null}$*;*
(3) $R = (\neg R_1)$ *and* $(R_1, DS)\theta = \varepsilon$*;*
(4) $R = (R_1 \vee R_2)$ *and* $(R_1\theta \neq \top \ \wedge \ R_2\theta \neq \top) \ \wedge \ (R_1\theta = \varepsilon \ \vee \ R_2\theta = \varepsilon)$*;*
(5) $R = (R1 \wedge R2)$ *and* $R_1\theta = \varepsilon \ \vee \ R_2\theta = \varepsilon$*.*

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

The semantics of OPT, as briefly mentioned in Section 2, requires special care in case the optional part consists of a graph pattern with a filter expression. Intuitively OPT can be considered as a form of an outer join. To consider only one case an optional pattern of the form $(P1 \ \mathsf{OPT} \ P2)$, which does not directly incorporate a FILTER in P_2 at the top level, will be viewed as $(P_1 \ \mathsf{OPT} \ (P_2 \ \mathsf{FILTER} \ R))$ with a R being the trivial FILTER, which is always true. Then we define the semantics as follows

Definition 2.10 (Evaluation of OPT with FILTERs (Harris & Seaborne, 2013, Section 18.5)) *Let* P_1, P_2 *be graph patterns and* R *a* FILTER *expression, then a mapping* θ *is in* $[[(P_1 \ \mathsf{OPT} \ (P_2 \ \mathsf{FILTER} \ R))]]_{DS}$ *if and only if:*

- $\theta = \theta_1 \cup \theta_2$*, s.t.* $\theta_1 \in [[P_1]]_G$*,* $\theta_2 \in [[P_2]]_{DS}$ *are compatible and* θ ***satisfies*** R*, or*
- $\theta \in [[P_1]]_{DS}$ *and there is no compatible* $\theta_2 \in [[P_2]]_{DS}$ *for* θ*, or*
- $\theta \in [[P_1]]_{DS}$ *and there exists a* $\theta_2 \in [[P_2]]_{DS}$*, s.t.* θ *and* θ_2 *are compatible and* $\theta \cup \theta_2$ ***does not satisfy*** R*.*

In the base case, if R is always true, then the semantics coincides with the outer join since we can omit the third condition, i.e. $[[P_1 \ \mathsf{OPT} \ P_2]]_{DS} = [[P_1]]_{DS} \text{⟕} [[P_2]]_{DS}$. Overall this special handling is required, since P_2 is allowed to exclude variables occurring in $vars(R)$, which are not considered unbound if they occur in P_1.

We now turn our attention to the semantics of the new features of SPARQL1.1. We begin with the semantics of solution modifiers. We note that the solution modifiers operate on ordered lists of solutions (with duplicates) instead of multisets of solutions. That means that before solution modifiers are applied, the solutions of the underlying patterns must be converted to lists. [14]

[14] In the specification this conversion is done by an explicit function toList, while here we encode this directly in the operator *order*.

We will use L as the notation for an ordered list. Let the elements contained in a list L be accessible by $L[i]$ with i starting from 1 up to the number of elements in L. The usual set theoretic notation also applies here for membership in the list, i.e. $\{\!\{\theta \in L\}\!\}$ is the multiset containing all elements in L. If elements are removed from this list, we assume that this data structure fills the gaps automatically and the index i always starts with 1 up to the number of elements in L. Additionally removing elements preserves the order on the remaining ones. Further let $|L|$ denote the number of elements in L. We also define an auxiliary concept of restricting a substitution θ to a set of variables $S \subseteq Var$ denoted as $\theta_{|S}$, which is the mapping such that $x\theta_{|S} = x\theta$ for $x \in S$ and $dom(\theta_{|S}) = dom(\theta) \cap S$.

Definition 2.11 (Solution modifiers) *Let Ω be a multiset of substitutions, L and L' lists, $\vec{V}$ a list of variables in Var and $S \subseteq Var$. Then we define the following basic operators*

$$
\begin{aligned}
&\textit{order}(\Omega, \vec{V}) = L \text{ where } L \text{ contains all elements of } \Omega \text{ ordered lexicographically by } \vec{V}\\
&\pi(S, L) \quad = L' \text{ where } L'[i] = L[i]_{|S} \text{ for } 1 \leq i \leq |L|\\
&\textit{dst}(L) \quad = L' \text{ where } L' \text{ is obtained by removing from } L \text{ every } L[i] \text{ where } \exists j < i \text{ such}\\
&\qquad\qquad \text{that } L[i] = L[j]\\
&\textit{lmt}(L, l, o) \;= L' \text{ where } L' \text{ is obtained by removing from } L \text{ all } L[i] \text{ with } i \leq o \text{ or } o + l < i
\end{aligned}
$$

Here, $\textit{order}(\Omega, \vec{V})$ is a function which orders the multiset by the values of bindings for $\vec{V} = (v_1, v_2, \ldots)$ lexicographically. i.e. first ordering substitutions in Ω according to bindings to v_1, then ordering according to bindings to v_2 within the substitutions binding to the same value for v_1, and so on. Values are ordered following the precedence order defined in (Harris & Seaborne, 2013, Section 15.1). Without loss of generality, and in order to always obtain deterministic orderings we assume $\vec{V}$ to contain any variables that are bound in any of the substitutions in Ω.

The remaining solution modifiers act in a straightforward manner. The projection operator π projects the variables of the set S, *dst* discards duplicates and finally *lmt* applies LIMIT and OFFSET, by discarding all substitutions outside the range given by o and $o + l$.

Note that due to the defined order in which solution modifiers are applied ($order \rightarrow \pi \rightarrow dst \rightarrow lmt$) according to the specification (Harris & Seaborne, 2013, Section 18.5), it plays a role for the lmt operator which of the duplicates is preserved by the dst operator within a list of solutions. However, the specification does not actually specify which of the duplicates are preserved for DISTINCT queries, leaving this aspect up to implementations; herein, we decided to retain those duplicate which are ordered first.

We are now ready to define the semantics of a subquery, which is just the evaluation of its graph pattern and then applying the solution modifiers afterwards on the result. The sequence in which the solution modifiers are applied is exactly specified (Harris & Seaborne, 2013, Section 15). We first order the substitutions, then project, followed by discarding duplicates and finally apply the slice of LIMIT and OFFSET. Note that the ordering is discarded after a subquery is evaluated. Only the final ordering of the overall query is shown in the result.

Definition 2.12 (Evaluation of subqueries) *Let $Q = (P, SM)$ be a subquery and $SM = (\vec{V}, S, dst, l, o)$ a solution modifier as above, then the evaluation $[[\cdot]]$ is defined as follows:*

$$[[Q]]_{DS} = \{\!\{\theta \in [[SM(P)]]_{DS}\}\!\}$$
$$[[SM(P)]]_{DS} = \begin{cases} \mathit{lmt}(\mathit{dst}(\pi(S, \mathit{order}([[P]]_{DS}, \vec{V}))), l, o)) \text{ if } dst = true \\ \mathit{lmt}(\pi(S, \mathit{order}([[P]]_{DS}, \vec{V})), l, o)) \text{ if } dst = false \end{cases}$$

Property path patterns without variable length parts can be defined via the graph patterns AND and UNION and a fresh variable for the join. Paths with arbitrary length are defined with the help of the ALP function (Harris & Seaborne, 2013, Section 18.4), which just starts at the given node and iteratively extends the solution set ($reachable$) with one application of the given property path. Note that the starting term is also reachable, which is required for the zero-length paths.

Algorithm 1: ALP$(a, PP, reachable, DS)$

input : $a \in I \cup B \cup L$, PP a property path, $reachable \subseteq I \cup B \cup L$, DS a dataset
output: $reachable$ containing all nodes reachable from a via PP
if $a \in reachable$ **then**
  return $\emptyset$
end
$reachable := reachable \cup \{a\}$;
$\Omega := [[(a, PP, ?X)]]_{DS}$;
foreach $b = \theta(?X), \theta \in \Omega$ **do**
  $reachable := reachable \cup \mathsf{ALP}(b, PP, reachable, DS)$;
end
return $reachable$

Based on the ALP function the property path patterns can now be defined as follows.

Definition 2.13 (Evaluation of property path patterns) *Let $s, o \in I \cup L \cup Var$, $N, N' \subseteq I$, $S = vars(\{s, o\})$, x a fresh variable, $DS = (G, G_{named})$ a dataset and PP, PP_1, PP_2 property paths, then the evaluation $[[\cdot]]_{DS}$ is defined as follows:*

$$\begin{aligned}
[[(s, PP_1/PP_2, o)]]_{DS} &= \{\!\{\theta_{|S} \mid \theta \in [[((s, PP_1, x)\ \mathit{AND}\ (x, PP_2, o))]]_{DS})\}\!\} \\
[[(s, PP_1|PP_2, o)]]_{DS} &= [[((s, PP_1, o)\ \mathit{UNION}\ (s, PP_2, o))]]_{DS} \\
[[(s, \hat{}PP, o)]]_{DS} &= [[(o, PP, s)]]_{DS} \\
[[(s, !(N, N'), o)]]_{DS} &= [[((s, !(N), o)\ \mathit{UNION}\ (o, !(N'), s))]]_{DS} \\
[[(s, !(N), o)]]_{DS} &= \{\!\{\theta_{|S} \mid dom(\theta) = S \cup \{x\} \text{ and } (s, x, o)\theta \in G \text{ with } x\theta \notin N\}\!\} \\
[[(s, PP?, o)]]_{DS} &= \{\theta \mid \theta \in [[(s, PP, o)]]_{DS}\} \cup \{\theta \mid s\theta = o\theta, dom(\theta) = S, \text{ and } s\theta \in G\} \\
[[(s, PP+, o)]]_{DS} &= \{\theta \mid \theta' \in [[(s\theta, PP, x)]]_{DS}, dom(\theta) = S, \text{ and} \\
&\quad\ o\theta \in \mathit{ALP}(x\theta', PP, \emptyset, DS)\} \\
[[(s, PP^*, o)]]_{DS} &= \{\theta \mid o\theta \in \mathit{ALP}(s\theta, PP, \emptyset, DS), dom(\theta) = S, \text{ and } s\theta, o\theta \in G\}\ ^{\dagger}
\end{aligned}$$

For zero-or-more length paths we search for substitutions, which start at $s\theta$ and go through arbitrarily many times the application of PP to reach $o\theta$. Additionally we have to make sure that

[†]We collect for the negated path in separate sets the normal and inverse IRIs: normal IRIs in N and the latter in N'. Note that the order is not relevant, see (Harris & Seaborne, 2013), Sections 9.1 and 18.2.2.

both $s\theta$ and $o\theta$ are in G, since otherwise, if both are variables we would not have a bound in them inside the graph. For the special case that s and o are both not variables the condition that both must be in G can be dropped from the definitions of the zero-or-more and zero-or-one paths.

The introduction of the MINUS pattern is straightforward, but we cannot reuse the relational operator for difference, "$-$", directly due to a slightly different definition (Harris & Seaborne, 2013, Section 18.5).

Definition 2.14 (MINUS graph pattern) *Le P_1, P_2 be graph patterns and $DS = (G, G_{named})$ a dataset, then the evaluation $[[P_1 \textsf{ MINUS } P_2]]_{DS}$ is defined as follows:*

$$[[P_1 \textsf{ MINUS } P_2]]_{DS} = \{\!\{\theta_1 \in [[P_1]]_{DS} \mid \text{for all } \theta_2 \in [[P_2]]_{DS},\ \theta_1 \text{ and } \theta_2 \text{ not compatible or } dom(\theta_1) \cap dom(\theta_2) = \emptyset\}\!\}$$

Additionally to the compatibility check we make sure that the substitutions do not have a disjoint domain. This would imply compatible substitutions and e.g. a single empty substitution in $[[P_2]]_{DS}$ would remove all solutions from $[[P_1]]_{DS}$.

The other type of newly introduced negation is a new atomic expression of the form $R = \textsf{EXISTS}(P)$. Then $R\theta = \top$ for a substitution θ if $[[P\theta]]_{DS} \neq \emptyset$ and $\bot$ else. By $P\theta$ we mean the natural generalization of the variable replacement on triples, by just replacing every variable of $dom(\theta)$ in the whole pattern P by the value given by θ. Note that there are some special cases, where such a substitution does not occur, i.e. variables in VALUES patterns and the assigned variable in BIND patterns. The dataset DS is taken from $[[P \textsf{ FILTER } R']]_{DS}$ where R occurs in R'.

Lastly let us define the semantics of the two assignment patterns. The VALUES pattern is straightforward and just specifies a table of new substitutions. For the BIND pattern we make use of an $eval(e)$ oracle function, which computes an expression e. As normal, an application of a substitution on the expression ($e\theta$) just replaces all variables by a term. Note that here, similarly to FILTER expressions, errors may occur. An expression which outputs an error is left unbounded. For this we make use of the *extend* function for error handling.

$$extend(\theta, x, e) = \begin{cases} \theta \cup [x \to eval(e\theta)] \text{ if } eval(e\theta) \neq \varepsilon \text{ and } x \notin dom(\theta) \\ \theta \text{ if } eval(e\theta) = \varepsilon \text{ and } x \notin dom(\theta) \\ \text{undefined else} \end{cases}$$

Definition 2.15 (Evaluation of assignment patterns) *Let P be a graph pattern, $DS = (G, G_{named})$ a dataset, $\vec{V} \in Var^n$ a list of variables and D a list of tuples of arity n and e an expression, then the evaluation $[[\cdot]]_{DS}$ is defined as follows:*

$$[[P \textsf{ BIND } e \textsf{ AS } x]]_{DS} = \{\!\{extend(\theta, x, e) \mid \theta \in [[P]]_{DS}\}\!\}$$
$$[[\textsf{VALUES } \vec{V}\ D]]_{DS} = \{\!\{\theta \mid dom(\theta) = \vec{V} \text{ and } \vec{V}\theta \in D\}\!\}$$

3 Datalog, Answer Sets, and External Predicates

In this paper we will use a very general form of Datalog commonly referred to as Answer-Set Programming (ASP), i.e. function-free logic programming (LP) under the answer-set semantics, which is an extension of the stable model semantics (Gelfond & Lifschitz, 1988, 1991). ASP is widely proposed as a useful tool for various problem solving tasks in e.g. Knowledge Representation and Deductive databases; various overviews of ASP as well as text books are available (Marek, 1999; Niemelä, 1999; Lifschitz, 1999; Baral, 2003; Brewka, Eiter, & Truszczynski, 2011; Gebser, Kaminski, Kaufmann, & Schaub, 2012). ASP extends Datalog with useful features such as default negation and disjunction in rule heads (Gelfond & Lifschitz, 1991); other extensions include external predicates (Eiter, Ianni, Schindlauer, & Tompits, 2005), aggregates (Faber, Leone, & Pfeifer, 2004), choice rules, cardinality and weight constraints (Niemelä, Simons, & Soininen, 1999), etc. Herein, we use a simplified version of ASP (we do not consider disjunction in rule heads, nor strong negation) with external predicates, borrowing definitions from so-called HEX-programs (Eiter et al., 2005).[15]

Let $Pred$, $Const$, Var, $exPr$ be sets of predicate names, constants, variable symbols, and external predicate names, respectively. Note that we assume all these sets except $Pred$ and $Const$ (which may overlap), to be disjoint. In accordance with common notation in LP and the notation for external predicates from (Eiter et al., 2006) we will in the following assume that $Const$ and $Pred$ comprise the (infinite) sets of numeric constants, string constants beginning with a lower case letter, or '"' quoted strings. Var is the set of string constants beginning with an upper case letter. Given $p \in Pred$ a (regular) *atom* is defined as $p(t_1, \ldots, t_n)$, where n is called the arity of p and $t_1, \ldots, t_n \in Const \cup Var$.

An *external atom* is of the form

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

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input list* and *output list*, respectively), and $g \in exPr$ is an external predicate name. We assume that for $g \in exPr$ the in- and output lists have fixed lengths $in(g) = n$ and $out(g) = m$, respectively. 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 constants.

Example 3.1 *In our translation, we will use the external predicate* `rdf`*, which intuitively takes as an input an IRI i and returns the RDF triples of the RDF graph which is accessibly at IRI i; that is, for* $\texttt{rdf}[i](s, p, o)$ *$i \in Const \cup Var$ is an input term, whereas $s, p, o \in Const \cup Var$ are output terms that may be bound by the external predicate and the external atom $rdf[i](s, p, o)$ is true if (s, p, o) is an RDF triple in the RDF graph which is accessibly at IRI i. We will use this external predicate in our translation to construct a dataset (cf. 2.1) comprising various RDF graphs.*

[15] In fact, the external predicates we use herein do not require the higher-order capabilities of HEX-programs, but the alternative form of external predicates defined in (Calimeri & Ianni, 2005) would likewise suffice for our purposes; we still stick with the notation of HEX-programs.

In the translation of the BIND pattern in Section 4.4 and subsequently by CONSTRUCT queries in Section 4.5.3 we will utilize further external predicates as needed.

Definition 3.1 (Rule) *Finally, a* rule *is of the form*

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

where h and b_i ($1 \leq i \leq n$) are regular or external atoms and not *is the symbol for default negation. If $n = 0$, then r is called a* fact.

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 a rule 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 a binding pattern. More precisely, we assume the following condition which extends the standard notion of safety (cf. (Ullman, 1989)) in Datalog with negation.

Definition 3.2 (Safety) *We call a rule* safe*, if any variable appearing in a rule appears in a non-default-negated regular body atom or as an output term of a non-default-negated external atom.*

Definition 3.3 (Program) *An ASP* program Π *is defined as a set of safe rules r of the form (1).*

We borrow the definition of the semantics of ASP programs from the more general HEX-programs (Eiter et al., 2005) which generalizes the answer-set semantics (Gelfond & Lifschitz, 1991), and is defined using the *FLP-reduct* (Faber et al., 2004) instead of the traditional Gelfond-Lifschitz reduct from (Gelfond & Lifschitz, 1991).

The *Herbrand base* of a program Π, denoted HB_Π, is the set of all possible ground versions of atoms and external atoms occurring in Π obtained by replacing variables with constants from $Const$. The grounding of a rule r, $ground(r)$, is defined accordingly, and the grounding of program P is $ground(P) = \bigcup_{r \in P} ground(r)$.

An *interpretation relative to* Π is any subset $\mathcal{I} \subseteq HB_\Pi$ containing only regular atoms. We say that $\mathcal{I}$ is a *model* of an atom $a \in HB_\Pi$, denoted by $\mathcal{I} \models a$, if $a \in \mathcal{I}$. With every external predicate name $g \in exPr$ with arity n we associate an $(n+1)$-ary Boolean function f_g (called *oracle function*) assigning each tuple $(\mathcal{I}, t_1 \ldots, t_n)$ either 0 or 1. [16] We say that $\mathcal{I} \subseteq HB_\Pi$ is a *model* of a ground external atom $a = g[t_1, \ldots, t_m](t_{m+1}, \ldots, t_n)$, denoted $\mathcal{I} \models a$, if and only if $f_g(\mathcal{I}, t_1, \ldots, t_n) = 1$.

Definition 3.4 (Model) *Let r be a ground rule. We define*

(i) $\mathcal{I} \models B(r)$ if and only if $\mathcal{I} \models a$ for all $a \in B^+(r)$ and $\mathcal{I} \not\models a$ for all $a \in B^-(r)$, and

(ii) $\mathcal{I} \models r$ if and only if $I \models H(r)$ whenever $\mathcal{I} \models B(r)$.

[16] This general notion of an oracle function reflects the intuition that external predicates compute (sets of) outputs for a particular input, depending on the interpretation. The dependence on the interpretation is necessary for instance for defining the semantics of external predicates querying OWL (Eiter et al., 2005) or computing aggregate functions (Polleres et al., 2007).

We say that $\mathcal{I}$ is a model *of a program Π, denoted $\mathcal{I} \models \Pi$, if and only if $\mathcal{I} \models r$ for all $r \in ground(\Pi)$.*

We define answer sets accordingly.

Definition 3.5 (Answer set) *The* FLP-reduct *(Faber et al., 2004) of Π with respect to $\mathcal{I} \subseteq HB_\Pi$, denoted $\Pi^\mathcal{I}$, is the set of all $r \in ground(\Pi)$ such that $\mathcal{I} \models B(r)$. $\mathcal{I} \subseteq HB_\Pi$ is an* answer set of Π *if and only if $\mathcal{I}$ is a minimal model of $\Pi^\mathcal{I}$. By $as(P)$ we denote the set of all answer sets of P.*

We do not consider further extensions common to many ASP dialects here, namely disjunctive rule heads, strong negation (Gelfond & Lifschitz, 1991). We do, however, assume a built-in predicate '$<$' for ordering constants, which is commonly found in ASP solvers nowadays. For our purposes we assume that this ordering predicate adheres to the order in the SPARQL specification (Harris & Seaborne, 2013, Section 15.1), i.e. it orders terms accordingly even if they are of a different type from lowest to highest:

1) null

2) Blank nodes

3) IRIs

4) RDF literals

Inside these classes the ordering is defined by lexicographically ordering the strings.

3.1 Restricted classes of ASP programs

We will call a program without external atoms a *pure* ASP program, and denote the classes of such pure programs and ASP programs with external atoms by ASP and ASP_{ex}, respectively; obviously $ASP \subset ASP_{ex}$. Analogously, we call the class of positive programs without default negation ASP_{pos} (and $ASP_{pos,ex}$, respectively). We define further restricted classes of ASP programs by means of the so called-dependency graph of the program:

Definition 3.6 (Dependency graph, ground dependency graph) *The* dependency graph Dep_Π *of a program Π is a directed labeled graph where each predicate name in $Pred \cup exPr$ occurring in Π is a node and (i) there is an unlabeled edge from l' to l if there is a rule r in Π such that $l \in H(r)$ and $l' \in B^+(r)$; (ii) there is a* `not` *-labeled edge from l' to l if there is a rule r in Π such that $l \in H(r)$ and $l' \in B^-(r)$.*

Likewise, we can define a ground dependency graph $Dep_{ground(\Pi)}$ *where analogous edges are defined with respect to the rules in $ground(\Pi)$ between elements of HB_Π (instead of between predicates).*

We call a program Π

- *non-recursive* if Dep_Π is acyclic,
- *stratified*, if Dep_Π does not contain cycles that involve any `not` -labeled edges, and
- *locally stratified*, if $Dep_{ground(\Pi)}$ does not contain cycles that involve any `not` -labeled edges.

Accordingly, we denote the classes of non-recursive, stratified and locally stratified programs by ASP^{nr}, ASP^{strat}, and ASP^{lstrat}, respectively. Let us illustrate the following obvious inclusion relations between the syntactic fragments of ASP:

$$ASP^{nr}_{pos} \subset ASP^{nr} \subset ASP^{strat} \subset ASP^{lstrat} \subset ASP$$
$$ASP_{pos} \subset ASP^{strat} \subset ASP^{lstrat} \subset ASP$$

For a detailed discussion about the implications of adding recursion and particularly non-stratified negation on expressive power and complexity, we refer to (Dantsin, Eiter, Gottlob, & Voronkov, 2001) and references therein. Here, we only note that for both non-recursive programs as well as (locally) stratified programs the answer set is always unique and finite.

As widely known, for programs without external predicates, safety as defined above guarantees the size of minimal models, i.e. answer sets) is finite. However, by external atoms in rule bodies, new, possibly infinitely many, ground atoms could be generated, even if all atoms themselves are safe in case of recursive programs. In order to avoid this, a stronger notion of safety for programs is defined in (Schindlauer, 2006): Without going into detail, informally, this notion says that a program is *strongly safe*, if no external predicate recursively depends on itself, thus defining a notion similar in spirit to (local) stratification over external predicates. Strong safety guarantees finiteness of models as well as finite computability even in the presence of external atoms.

We will get back to which fragment of ASP is needed for which parts of our translation in more detail in Section 5.

4 From SPARQL to ASP

Having defined the formal semantics of SPARQL queries, we are now ready to define a translation from SPARQL to the fragments of ASP in the previous section.

We start with a core translation, encoding many important concepts and features of SPARQL. Afterwards we will show how to encode the newly available query components of SPARQL1.1 in ASP. Particularly, for BIND patterns we will make use of external atoms, since new values may be introduced by this pattern, which also requires several changes in other parts of the translation, which we will discuss separately.

4.1 Core Translation τ

The core translation encodes graph patterns and FILTER expressions by recursively walking through the parse tree of a given query. For the algebraic version of SPARQL, the parse tree of a query is simply a tree consisting of the subpatterns of a given pattern.

Definition 4.1 (Subpattern) *A graph pattern P' is a subpattern of a graph pattern P if P' occurs in P as a substring. A subpattern P' is an immediate subpattern of P if there is no graph pattern P'' such that P'' is a subpattern of P and P' is a subpattern of P''.*

Using subpatterns we can straightforwardly define a parse tree of a pattern.

Definition 4.2 (Parse tree) *A rooted tree $T = (V, E)$ is called a parse tree of a graph pattern P if*

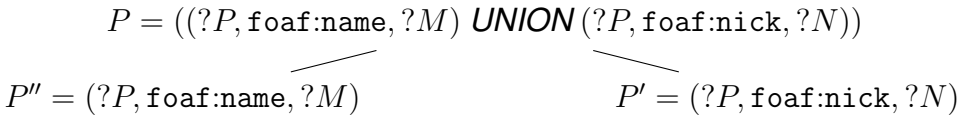
- *$V = \{P' \mid P'$ is a subpattern of $P\}$,*
- *P is the root of T and*
- *$(P_1, P_2) \in E$ iff P_2 is an immediate subpattern of P_1.*

We will associate integers with each vertex of the parse tree of a graph pattern P, starting with 1 for the pattern itself and the root vertex. Note that in all cases a vertex in the parse tree has at most two children, due to the definition of the SPARQL graph patterns. Hence we will number the vertices recursively by simply numbering children of a vertex with number i with the numbers $2 * i$ and $2 * i + 1$. To gain deterministic results we number the vertex with the subpattern occurring left in the pattern of the $i - th$ vertex with the lower integer. For a pattern occurring the parse tree, we will call the associated integer its position in the parse tree. This identifier will later be used for unique names of ASP predicates.

Likewise we can define such a parse tree for filter expressions. In our translation parse trees will be used implicitly by recursively decomposing a given pattern in the same manner as in the tree representation.

We will now first start with an intuitive introduction to our translation. Given a pattern, every subpattern is translated to a set of rules. The main predicate for the translation is called $\texttt{ans}_i$ for the parse tree at position i. This predicate acts as a representation of the computed substitutions and hence has variable arity depending on the current graph pattern of the query. Again the distinct constant null is used for unbound variables.

Example 4.1 *Consider the pattern P of Example 2.6, without solution modifiers, i.e. $P = ((?P, \texttt{foaf:name}, ?M)$ UNION $(?P, \texttt{foaf:nick}, ?N))$. To keep things simple let this be a DISTINCT query. The parse tree of P is given as follows.*



The main part of the translation constructs now rules for each pattern in the tree.

```
ans1(M,null,P,default) :- ans2(M,P,default).
ans1(null,N,P,default) :- ans3(N,P,default).
ans2(M,P,default) :- triple(P,foaf:name,M,default).
ans3(N,P,default) :- triple(P,foaf:nick,N,default).
```

Here we see several parts of the translation. We have a main predicate for each pattern in the tree, e.g. $\texttt{ans}_i$ *for* $i \in \{1, 2, 3\}$ *in our case. Intuitively, the rules for the patterns at position* 2 *and* 3 *just extract the right triples. Of importance is that these predicates directly correspond to substitutions. Note that the constant* $default$ *is used to represent the default graph.*

The complex pattern for UNION *is translated by two rules, which both compute solutions for this pattern. The predicate* $\texttt{ans}_1$ *represents substitutions for the variables* $?M$*,* $?N$ *and* $?P$*. Since, e.g.* $?N$ *does not occur in the subpattern* P'*, we directly use* null *as the second term of this predicate, which is our special constant for unbound variables. So intuitively a ground atom* $\texttt{ans}_1(t_1, t_2, t_3, default)$ *can be read as substitutions for the three variables, where the value for* $?M$ *is given by* t_1*,* $?N$ *by* t_2 *and* $?P$ *by* t_3*. Obviously for a working translation the ordering for this is crucial and can be extracted first by looking at the variables of the pattern and then ordering them lexicographically. For the full translation, these rules need to be augmented by further ASP modules, which import all the triple patterns, as well as adding some auxiliary rules.*

Let P be a graph pattern. Without loss of generality, we assume that no variable name in P starts with U or V. We will use these prefixes to introduce new variables for UNION patterns (U) and in auxiliary rules added for FILTER evaluations (V). As before we define $vars(P)$ to be the set of all variables occurring in P excluding filters. We need to distinguish auxiliary variables introduced in different parts of the parse tree, hence we define an indexed set of variables, which includes the necessary variables for our translation, as follows.

Definition 4.3 (Indexed variable set) *Given a positive integer* i*,* $dst \in \{true, false\}$ *and a graph pattern* P*, we define the indexed variable set* $vars_i^{dst}(P)$ *as follows:*

- *For* $P = (s, p, o)$ *we define* $vars_i^{dst}(P) = vars(P)$.
- *For* $P = (P_1$ AND $P_2)$ *or* $(P_1$ OPT $P_2)$*, let* $vars_i^{dst}(P) = vars_{2*i}^{dst}(P_1) \cup vars_{2*i+1}^{dst}(P_2)$.
- *For* $P = (P_1$ FILTER $R)$*, let* $vars_i^{dst}(P) = vars_{2*i}^{dst}(P_1)$.
- *For* $P = ($GRAPH $g\ P_1)$*, let* $vars_i^{dst}(P) = vars_{2*i}^{dst}(P_1) \cup \{g \mid g \in Var\}$
- *For* $P = (P_1$ UNION $P_2)$*, let* $vars_i^{true}(P) = vars_{2*i}^{true}(P_1) \cup vars_{2*i+1}^{true}(P_2)$.
- *For* $P = (P_1$ UNION $P_2)$*, let* $vars_i^{false}(P) = vars_{2*i}^{false}(P_1) \cup vars_{2*i+1}^{false}(P_2) \cup \{\texttt{U}_i\}$.

That is, for each UNION pattern, its indexed variable set introduces a new, auxiliary variable. The idea here is that the index assigned to this fresh variable is a unique index identifying the node of the pattern in the parse tree of the graph pattern. This auxiliary variable is required for preserving duplicates, thus we omit it if dst is set to true. We note that in this definition, $vars_i^{true}(P) = vars(P)$, but this will change when we extend this function for SPARQL1.1 features in Section 4.3. Intuitively speaking, if dst is true then $vars_i^{true}(P)$ contains all variables of the pattern P at position i, which may be mapped by a substitution in $[[P]]_{DS}$.

Recall that by $\overline{V}$ we denote the tuple obtained from lexicographically ordering a set of variables in V. The notion $\overline{V}[V' \to c]$ means that, after ordering V all variables from $V \cap V'$ are replaced

by constant c. We also occasionally write short $\overline{V}[v \to c]$ in case that $V' = \{v\}$ is a singleton set, and write $\overline{V}[V' \to c_1, V'' \to c_2]$ for $(\overline{V}[V' \to c_1])[V'' \to c_2]$. Also sometimes we will use $\overline{V}'$ to denote the uniform renaming of all variables in V after ordering by appending a prime. If we only rename a subset $X \subseteq V$, then we will use $\overline{V}[X \to X']$.

We use a similar notion for replacing variables in FILTER expressions, i.e., if R is a FILTER expression, then $R[V \to c]$ is the expression obtained from replacing all occurrences of variables in V within R by constant c.

Let $Q = (DS, P, SM)$, where $DS = (G, G_{named})$ as defined in Section 2. We will define the core translation function $\tau(P, dst, D, NodeIndex)$ which takes a pattern P, a graph name $D \in \{default\} \cup G_{named}$, the boolean value $dst \in \{true, false\}$ to denote that the DISTINCT keyword is present and a positive integer as input, and results in a non-recursive Datalog program. The $NodeIndex$ parameter is being used to give a unique index to each sub-pattern and sub-FILTER-expression, which we use to generate unique predicate names for auxiliary predicates during our translation. The translation is defined recursively depending on the structure of P.

The semantics of SPARQL specifies that two substitutions are compatible if for each shared variable both substitutions agree or one of them is null. Obviously this notion of compatibility is not present in ASP per se, since only equal terms may be unified. To cope with this we will utilize helper predicates $\texttt{join}_n$, which make sure that null joins with every value. In the core translation only the AND and OPT encoding require joining over null values. We will outline first the simple mechanism behind this. Assume that we are given the pattern $P = (P_1 \text{ AND } P_2)$ at position i. Then the following rule simulates the AND by joining $\texttt{ans}_{2*\texttt{i}}(\overline{vars^{dst}_{2*i}(P_1)}, D)$ and $\texttt{ans}_{2*\texttt{i}+1}(\overline{vars^{dst}_{2*i+1}(P_2)}, D)$, the ASP predicates which compute the solutions of the subpatterns P_1 and P_2, over their shared variables if no null values are present:

$$\texttt{ans}_\texttt{i}(\overline{vars^{dst}_i(P)}, D) \ \texttt{:-}\ \texttt{ans}_{2*\texttt{i}}(\overline{vars^{dst}_{2*i}(P_1)}, D), \texttt{ans}_{2*\texttt{i}+1}(\overline{vars^{dst}_{2*i+1}(P_2)}, D).$$

A special handling is required for the shared variables of the two patterns, hence we define the concept of shared variables of two patterns as follows.

Definition 4.4 (Shared variables) *Let P_1 and P_2 be two graph patterns. The set of their shared variables is defined as $S_{P_{i_1},P_{i_2}} = vars^{true}_1(P_{i_1}) \cap vars^{true}_2(P_{i_2})$.*

In the core translation this means that $S_{P_{i_1},P_{i_2}} = vars(P_{i_1}) \cap vars(P_{i_2})$, but in the following we will also extend the translation with subqueries, which may project certain variables out inside a graph pattern. This is why we here look for the variables of the two patterns, excluding auxiliary variables and projected out variables (which is achieved by setting dst to true). The indices in this case are not relevant.

Now to achieve the right semantics for null values we rename in both predicates the shared variables to distinguish their value "before" we join them. This is handled by the auxiliary predicate $\texttt{join}_\texttt{n}$, which we define in the following module for a given integer n.

Let $Join(n) =$

$\texttt{join}(X, X, X) \ \texttt{:-}\ term(X).\quad \texttt{join}(\mathsf{null}, \mathsf{null}, \mathsf{null}).$
$\texttt{join}(X, \mathsf{null}, X) \ \texttt{:-}\ term(X).$
$\texttt{join}(\mathsf{null}, X, X) \ \texttt{:-}\ term(X).$
$\texttt{join}_1(X_1', X_1'', X_1) \ \texttt{:-}\ \texttt{join}(X_1', X_1'', X_1).$
$\texttt{join}_2(X_1', X_2', X_1'', X_2'', X_1, X_2) \ \texttt{:-}\ \texttt{join}_1(X_1', X_1'', X_1), \texttt{join}(X_2', X_2'', X_2).$
$\texttt{join}_3(X_1', X_2', X_3', X_1'', X_2'', X_3'', X_1, X_2, X_3) \ \texttt{:-}\ \texttt{join}_2(X_1', X_2', X_1'', X_2'', X_1, X_2), \texttt{join}(X_3', X_3'', X_3).$
$\vdots$
$\texttt{join}_n(X_1', \ldots, X_n', X_1'', \ldots, X_n'', X_1, \ldots, X_n) \ \texttt{:-}$
$\quad \texttt{join}_{n-1}(X_1', \ldots, X_{n-1}', X_1'', \ldots, X_{n-1}'', X_1, \ldots, X_{n-1}), \texttt{join}(X_n', X_n'', X_n).$

Intuitively this predicate derives for two compatible substitutions the value for each variable. Now we add the auxiliary predicate $\texttt{join}_{|S_{P_1,P_2}|}$ to the body of the rule for the join of two patterns. This helper predicate is used to specify that any value joins with null for the number $|S_{P_1,P_2}|$ of shared variables. This results now in the following rule for the AND pattern.

$$\begin{aligned}\texttt{ans}_i(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-}\ & \texttt{ans}_{2*i}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D),\\ & \texttt{ans}_{2*i+1}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D),\\ & \texttt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}).\end{aligned}$$

That means a shared variable x is renamed to x' in the first predicate and to x'' in the second one. The unprimed version is bound by the $\texttt{join}_{|S_{P_1,P_2}|}$ and is equal to x' if $x' = x''$ or x'' is null or on the other hand equal to x'' if x' is null. Note that in the following translation this mechanism for joining is omitted if there are no shared variables to join.

We are now ready to define the core translation. We begin with the translation of the dataset. Here we have two possibilities, namely importing the graphs by a preprocessing step or creating the triples by the external predicate `rdf`, which directly accesses an IRI for retrieving the graphs. For now we will stick to the preprocessing variant, since we want to restrict ourselves to plain Datalog in the core translation. This means, given the dataset $DS = (G, G_{named})$ for a query Q, we import the graph by

$$\begin{aligned}\Pi_{DS} = & \{\texttt{triple}(s, p, o, \texttt{default}). \mid (s, p, o) \text{ is entailed by the graph represented by } d \in G\}\\ & \cup \{\texttt{triple}(s, p, o, g). \mid (s, p, o) \text{ is entailed by the graph represented by } g \in G_{named}\}\end{aligned}$$

Now the translation function $\tau(P, G, NodeIndex)$ is defined as follows.

(1) Let $P = (s, p, o)$ then $\tau(P, dst, D, i)$ is defined as

$$\texttt{ans}_i(\overline{vars_i^{dst}((s, p, o))}, D) \ \texttt{:-}\ \texttt{triple}(s, p, o, D).$$

(2) Let $P = (P_1 \ \mathsf{AND}\ P_2)$ then $\tau(P, dst, D, i) \ =$
$\tau(P_1, dst, D, 2 * i) \ \cup \ \tau(P_2, dst, D, 2 * i + 1) \cup$

$$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-}\ \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D),$$
$$\texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D),$$
$$\texttt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}).$$

(3) Let $P = (P_1 \ \mathsf{UNION}\ P_2)$ then $\tau(P, false, D, i) \ =$
$\tau(P_1, false, D, 2*i) \ \cup \ \tau(P_2, false, D, 2*i+1) \cup$

$$\texttt{ans}_\texttt{i}(\overline{vars_i^{false}(P)}[\texttt{U}_i \to 1, (vars_i^{false}(P) \setminus vars_{2*i}^{false}(P)) \to \mathsf{null}], D) \ \texttt{:-}\ \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{false}(P_1)}, D).$$
$$\texttt{ans}_\texttt{i}(\overline{vars_i^{false}(P)}[\texttt{U}_i \to 2, (vars_i^{false}(P) \setminus vars_{2*i+1}^{false}(P)) \to \mathsf{null}], D) \ \texttt{:-}\ \texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{false}(P_2)}, D).$$

Let $P = (P_1 \ \mathsf{UNION}\ P_2)$ then $\tau(P, true, D, i) \ =$
$\tau(P_1, true, D, 2*i) \ \cup \ \tau(P_2, true, D, 2*i+1) \cup$

$$\texttt{ans}_\texttt{i}(\overline{vars_i^{true}(P)}[(vars_i^{true}(P) \setminus vars_{2*i}^{true}(P)) \to \mathsf{null}], D) \ \texttt{:-}\ \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}(P_1)}, D).$$
$$\texttt{ans}_\texttt{i}(\overline{vars_i^{true}(P)}[(vars_i^{true}(P) \setminus vars_{2*i+1}^{true}(P)) \to \mathsf{null}], D) \ \texttt{:-}\ \texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{true}(P_2)}, D).$$

Note that $\texttt{U}_i$ serves here just for the purpose of preserving duplicates, i.e. if a substitution is obtained by one branch of the UNION pattern it is assigned either 1 or 2, depending on which branch it came from. This variable is not introduced in the case that $dst = true$.

(4) Let $P = (\mathsf{GRAPH}\ g\ P_1)$, $g \in V \cup I$, then $\tau(P, dst, D, i) \ = \ \tau(P_1, dst, g, i) \cup$
$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-}\ \texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P_1)}, g), \ \texttt{named}(g).$

We define a fixed rule set $\Pi^{DS}_{\mathsf{FILTER}}$ for correct FILTER processing. Note that the facts for IRIs, blank nodes and literals can be achieved by a preprocessing step, like the import of the RDF triples.

```
named(g).        for all g ∈ G_named
iri(i).          for all i ∈ I that appear in DS
blank(b).        for all b ∈ B that appear in DS
literal(l).      for all l ∈ L that appear in DS
null(null).
term(X)        :- iri(X).
term(X)        :- blank(X).
term(X)        :- literal(X).
equals(X,X,true)      :- term(X).
equals(X,Y,false)     :- term(X), term(Y), not equals(X,Y,true).
equals(null,Y,err)    :- term(Y).
equals(X,null,err)    :- term(X).
equals(null,null,err).
lowerThan(X,Y,true)      :- term(X), term(Y), X < Y.
lowerThan(X,Y,false)     :- term(X), term(Y), not lowerThan(X,Y,true).
lowerThan(null,Y,err)    :- term(Y).
lowerThan(X,null,err)    :- term(X).
```

```
lowerThan(null,null,err).
isIri(X,true)  :- iri(X).
isIri(X,false) :- blank(X).
isIri(X,false) :- literal(X).
isIri(null,err).
isLiteral(X,false) :- iri(X).
isLiteral(X,false) :- blank(X).
isLiteral(X,true)  :- literal(X).
isLiteral(null,err).
isBlank(X,false) :- iri(X).
isBlank(X,true)  :- blank(X).
isBlank(X,false) :- literal(X).
isBlank(null,err).
bound(X,true) :- term(X).
bound(X,false) :- null(X).
neg(true, false).      neg(false, true).      neg(err, err).
and(true,  true,  true).              or(true,  true,  true).
and(true,  false, false).             or(true,  false, true).
and(false, true,  false).             or(false, true,  true).
and(false, false, false).             or(false, false, false).
and(true,  err,   err).               or(true,  err,   true).
and(err,   true,  err).               or(err,   true,  true).
and(false, err,   false).             or(false, err,   err).
and(err,   false, false).             or(err,   false, err).
and(err,   err,   err).               or(err,   err,   err).
```

Now on basis of $\Pi^{DS}_{\mathsf{FILTER}}$ we can define the rules simulating filter evaluation. The basic idea is that atomic filter expressions are handled by $\Pi^{DS}_{\mathsf{FILTER}}$ and for complex filter expressions we just need to propagate the truth values in the correct manner. This propagation is handled by the last rules in $\Pi^{DS}_{\mathsf{FILTER}}$.

(5) Let $P = (P_1 \ \mathsf{FILTER}\ R')$, such that $R = R'[vars(R) \setminus vars_i^{true}(P) \to \mathsf{null}]$,
$\tau(P, dst, D, i) = \tau(P_1, dst, D, 2 * i) \cup$

$$\mathtt{ans_i}(\overline{vars_i^{dst}(P)}, D) \ \mathtt{:-}\ \mathtt{ans_{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D), Cond(R, true, i, 1).$$

$Cond(R, v, i, j)$, for $v \in \{true, false, err\} \cup Var$, and i, j positive integers, is defined recursively as follows.[17]

(5.1) if R is an atomic Filter, then $Cond(R, v, i, j) =$

- $\mathtt{equals}(X, Y, v)$, if R is $X = Y$.
- $\mathtt{lowerThan}(X, Y, v)$, if R is $X < Y$.
- $\mathtt{isIri}(X, v)$, if R is $\mathsf{isIRI}(X)$.
- $\mathtt{isLiteral}(X, v)$, if R is $\mathsf{isLITERAL}(X)$.

[17] Here, $true, false, err$ are special constants emulating the three truth values $\top, \bot, \varepsilon$ in Definition 2.9.

- $\texttt{isBlank}(X, v)$, if R is $\mathsf{isBLANK}(X)$.
- $\texttt{bound}(X, v)$, if R is $\mathsf{BOUND}(X)$.

(5.2) if $R = (\neg(R_1))$, then $Cond(R, v, i, j) = filter_{i,j}(\overline{vars(R)}, v)$, where the following additional rule is added to $\tau(P, D, i)$:

$$\texttt{filter}_{\texttt{i,j}}(\overline{vars(R)}, V) \ \texttt{:-} \ Cond(R_1, V_1, i, 2*j), neg(V_1, V).$$

(5.3) $R = (R_1 \wedge R_2)$, then $Cond(R, v, i, j) = filter_{i,j}(\overline{vars(R)}, v)$, where the following additional rule is added to $\tau(P, D, i)$:

$$\texttt{filter}_{\texttt{i,j}}(\overline{vars(R)}, V) \ \texttt{:-} \ Cond(R_1, V_1, i, 2*j), Cond(R_2, V_2, i, 2*j+1), and(V_1, V_2, V).$$

(5.4) $R = (R_1 \vee R_2)$, then $Cond(R, v, i, j) = filter_{i,j}(\overline{vars(R)}, v)$, where the following additional rules is added to $\tau(P, D, i)$:

$$\texttt{filter}_{\texttt{i,j}}(\overline{vars(R)}, V) \ \texttt{:-} \ Cond(R_1, V_1, i, 2*j), Cond(R_2, V_2, i, 2*j+1), or(V_1, V_2, V).$$

(6) Let $P = (P_1 \ \mathsf{OPT} \ (P_2 \ \mathsf{FILTER} \ R'))$. We define $R = R'[vars(R) \setminus (vars_i^{true}(P) \to \mathsf{null}]$.[18] Then, $\tau(P, dst, D, i) = \tau(P_1, dst, D, 2*i) \cup \tau(P_2, dst, D, 2*i+1) \cup$

$$\begin{aligned}
\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-} \ & \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D), \\
& \texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D), \\
& \texttt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}), Cond(R, true, i, 1).
\end{aligned}$$

$$\begin{aligned}
\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}[(vars_i^{dst}(P) \setminus vars_{2*i}^{dst}(P_1)) \to \mathsf{null}], D) \ \texttt{:-} \\
\texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D), \ \texttt{not ans}_\texttt{i}'(\overline{vars_i^{true}(P_1)}, D).
\end{aligned}$$

$$\begin{aligned}
\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}[(vars_i^{dst}(P) \setminus vars_{2*i}^{dst}(P_1)) \to \mathsf{null}, S_{P_1,P_2} \to S'_{P_1,P_2}], D) \ \texttt{:-} \\
& \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D), \\
& \texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D), \\
& \texttt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}), \texttt{not} \ Cond(R, true, i, 1).
\end{aligned}$$

$$\begin{aligned}
\texttt{ans}_\texttt{i}'(\overline{vars_i^{true}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D) \ \texttt{:-} \\
& \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D), \\
& \texttt{ans}_{\texttt{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D), \\
& \texttt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}).
\end{aligned}$$

For the translation of $(P_1 \ \mathsf{OPT} \ P_2)$ patterns without FILTER expressions we just append a filter which is always true, i.e. rewrite to $(P_1 \ \mathsf{OPT} \ (P_2 \ \mathsf{FILTER} \ R))$. This encoding directly matches the three conditions from Definition 2.10. Again we have to apply the joining over null helpers. In the last rule for $\texttt{ans}_\texttt{i}'$ we specify which substitutions are compatible to exclude them in the

[18] Here, we replace unsafe variables in the FILTER expressions by null.

second rule. Note that here we rename the variables in the head to not have to apply the join again in second rule. Further we can exclude all auxiliary variables by setting $dst = true$, since for checking compatibility it suffices to check once for a set of duplicate substitutions.

This completes the core translation. Let $Q = (DS, P)$ be a query without solution modifiers and $DS = (G, G_{named})$ as defined above. We translate this query to a logic program Π_Q defined as follows.

$$\Pi_Q = \Pi_{DS} \cup \tau(P, false, \texttt{default}, 1) \cup \Pi^{DS}_{\mathsf{FILTER}} \cup Join(n)$$

This logic program computes the query answer in $\texttt{ans}_1$ for n being the maximal cardinality of shared variables in rules in $\tau(P, \texttt{default}, 1)$. Note that to achieve a set-semantics instead of the multi-set semantics of the SPARQL specification, we just need to make sure that the dst flag is always set to true. In Section 4.2 we will explain how to extend this idea and encode solution modifiers in ASP.

We now show that the translation gives the correct result w.r.t. the formal semantics defined in Section 2.5. For this we first specify the relation between the answer-sets and the multi-set of substitutions of a given query and its translation.

Definition 4.5 (Correspondence) *Let P be a graph pattern, $i \geq 0$ an integer and $g_1, \ldots, g_n$ be ASP terms. We say that a substitution θ corresponds to a ground ASP atom $\mathrm{ans_i}(g_1, \ldots, g_n, default)$ w.r.t. the pair (P,i), in symbols $\theta \cong^{(P,i)} \mathrm{ans_i}(g_1, \ldots, g_n, default)$, if*

- *$dom(\theta) \subseteq vars_i^{true}(P)$ and*
- *for each $y_j \in vars_i^{true}(P)$, where $\overline{vars_i^{false}(P)} = (y_1, \ldots, y_n)$:*
  - *$y_j\theta = g_j$ if $y_j \in dom(\theta)$*
  - *$g_j = \mathsf{null}$ if $y_j \notin dom(\theta)$*

We say $[[P]]_{DS}$, with $DS = (G, G_{named})$ corresponds to an answer-set I w.r.t. the integer i, in symbols $[[P]]_{DS} \cong^i I$, for $n = |vars_i^{false}(P)|$, if

- *for each $\theta \in [[P]]_{DS}$ there is an atom $\mathrm{ans_i}(g_1, \ldots, g_n, default) \in I$, s.t. $\theta \cong^{(P,i)} \mathrm{ans_i}(g_1, \ldots, g_n, default)$,*
- *for each $\mathrm{ans_i}(g_1, \ldots, g_n, default) \in I$ there is a substitution $\theta \in [[P]]_{DS}$, s.t. $\theta \cong^{(P,i)} \mathrm{ans_i}(g_1, \ldots, g_n, default)$ and*
- *$|[[P]]_{DS}| = |\{\!\{\mathrm{ans_i}(g_1, \ldots, g_n, default) \in I\}\!\}|$.*

Recall that for a pattern P, the variable set $vars_i^{true}(P)$ contains all variables that may be mapped by a substitution in $[[P]]_{DS}$. Now a set of atoms of an answer-set corresponds to a multi-set of substitutions of a given pattern P and the position of P given by the integer i, if we can extract exactly the same substitutions from the $\mathrm{ans_i}$ predicate present in the set, as we have in the

multi-set of substitutions. Some of the substitutions in $[[P]]_{DS}$ might not have a value for a variable in $vars_i^{true}(P)$, which is then reflected by the null constant.

Next we show an easy Lemma, which states that we can use the $\mathrm{join_n}$ predicate for showing compatibility between substitutions.

Lemma 4.1 *Let P_1 and P_2 be two graph patterns, $n = |S_{P_1,P_2}|$, DS a dataset and $\theta_1 \in [[P_1]]_{DS}$ and $\theta_2 \in [[P_2]]_{DS}$ be two substitutions. If a stratified program Π contains the following rules:*

$$Join(n) \cup \{\texttt{term}(t). \mid x \in dom(\theta_1) \cup dom(\theta_2), x\theta_1 = t \text{ or } x\theta_2 = t\} \cup \{\texttt{null}(\mathsf{null}).\}$$

then θ_1 and θ_2 are compatible iff I is an answer-set of the program Π and the following atom is in I:

$$\begin{aligned}\mathrm{join}_n(&\overline{S_{P_1,P_2}}[x \to x\theta_1 \textit{ for } x \in dom(\theta_1), y \to \mathsf{null} \textit{ for } y \in S_{P_1,P_2} \setminus dom(\theta_1)],\\ &\overline{S_{P_1,P_2}}[x \to x\theta_2 \textit{ for } x \in dom(\theta_2), y \to \mathsf{null} \textit{ for } y \in S_{P_1,P_2} \setminus dom(\theta_2)],\\ &\overline{S_{P_1,P_2}}[x \to x(\theta_1 \cup \theta_2)])\end{aligned}$$

Proof: We show this by induction on $n = |S_{P_1,P_2}|$. The basic observation is that the join predicate with arity three is derived for all combinations of three constants, including null, s.t. if the first two terms are equal, the third term is also equal to them. Otherwise if one of the first two terms is null, then the third is equal to the other. Next, the grounded atom join_n has an arity of $3 * n$. In the first n terms we substitute all terms from the first substitution, or null if unbound. Likewise we do this for the second substitution. Note also that Π is a stratified program.

If $n = 1$, then join_1 computes exactly the same as join. Assume that V is the shared variable of both patterns and both substitutions have V in their domain. We then have the atom $\mathrm{join}_1(V\theta_1, V\theta_2, V(\theta_1 \cup \theta_2))$ is in the answer-set of Π iff both substitutions are compatible, since if they are, then either $V\theta_1 = V\theta_2$ and hence $V(\theta_1 \cup \theta_2) = V\theta_1$. If on the other hand, one of the substitutions do not map V to a value then this is handled by the constant null. Without loss of generality let $V \notin dom(\theta_1)$, then $\texttt{join}(\mathsf{null}, V\theta_2, V\theta_2)$ is derived. Similarly if both substitutions do not map V to a value. If both substitutions are not compatible, then $V\theta_1 \neq V\theta_2$ and both are not null, but such a corresponding atom is never derived in the answer-set of Π.

Assume the claim holds up to n, then to show that it holds also for $n + 1$ simply consider that just the $n + 1 - th$ variable must be included into our consideration, which is done by the base predicate join. □

We now show that our core translation is correct w.r.t. the formal semantics.

Proposition 4.2 *For a* SPARQL*1.0 graph pattern P and a dataset DS it holds that $[[P]]_{DS} \cong^1 I$ for I the answer-set of the program Π_Q with $Q = (DS, P)$.*

Proof: (sketch) Let P be a SPARQL1.0 graph pattern and DS a dataset. We construct the query $Q = (DS, P)$ and Π_Q. First note that Π_Q is a stratified answer-set program, hence it has a unique answer-set I.

We now show that for any subpattern P' of P at position i in the parse tree of P that $[[P']]_{DS} \cong^i I$ holds.

We show the correspondence by structural induction on the different graph patterns of the SPARQL1.0 specification used here, i.e. essentially using the parse tree of P. For simplifying the proof we will omit the GRAPH pattern, since this pattern changes the current graph name and complicates the structural induction. Nevertheless it can be shown straightforwardly to be correct w.r.t. formal semantics.

The *base case* is a triple pattern. Let $P' = (s, p, o)$. By Π_{DS} we imported the whole graph. Therefore the correspondence holds trivially, since if $\theta \in [[P']]_{DS}$ then there is a corresponding answer-set predicate $\mathrm{ans_i}$ for which the grounded versions of $\mathrm{ans_i}(\overline{vars(P')}, default)$ are present in I, due to the fact that if there is a match in the graph and hence a substitution in $[[P']]_{DS}$, then the atom is derived in the answer-set program, since the corresponding fact $\texttt{triple}(s, p, o, default)$ is present in the program. For the complex graph patterns we distinguish the different cases. For each we assume that the patterns at position $2 * i$ and $2 * i + 1$ are already shown to be in correspondence w.r.t. the ASP program and the formal semantics.

- If $P' = (P_1 \text{ AND } P_2)$, then for each $\theta \in [[P']]_{DS}$ there exists two compatible substitutions $\theta_1 \in [[P_1]]_{DS}$ and $\theta_2 \in [[P_2]]_{DS}$, s.t. $\theta = (\theta_1 \cup \theta_2)$. By induction hypothesis we assume that $[[P_1]]_{DS} \cong^{2*i} I$, likewise for the second pattern, $[[P_2]]_{DS} \cong^{2*i+1} I$. This means that $\theta_1 \cong^{(P_1, 2*i)} \mathrm{ans}_{2*\mathrm{i}}(g_1, ..., g_n, default)$ for some ground atom in I. This is likewise the case for θ_2. With Lemma 4.1 it is now straightforward to see that there is a $\theta \cong^{(P', i)} \mathrm{ans_i}(g'_1, ..., g'_n, default)$ with the latter being in I, since if θ_1 and θ_2 are compatible, then we derive the required $\mathrm{join}_{|\mathrm{S_{P_1,P_2}}|}$ predicate and in turn the corresponding predicate for θ for P' at position i. For any incompatible pairs we do not derive a new substitution for P', i.e. a new predicate corresponding to it, in our ASP program.

- If $P' = (P_1 \text{ FILTER } R)$, then we need to show that the filter expression R is handled correctly. Filter expressions have their own parse tree and we can again show the correspondence via structural induction. This is straightforward, since for all atomic filter expressions we derive via the auxiliary module Π^{DS}_{FILTER} all possible evaluations of such filter expressions w.r.t. all terms in the graphs. That is, we derive a predicate for each atomic filter expression for all terms in the graphs and their truth value. For complex filter expressions these truth values are propagated as defined for the three-valued logic.

- If $P' = (P_1 \text{ OPT } P_2)$, then the formal semantics specifies three cases, which are reflected in the translation.

  - The first case is that $\theta \in [[P']]_{DS}$ if $\theta = (\theta_1 \cup \theta_2)$ with $\theta_1 \in [[P_1]]_{DS}$ and $\theta_2 \in [[P_2]]_{DS}$ are compatible and $(\theta_1 \cup \theta_2)$ satisfies the filter, which is mapped directly by the first rule of the translation. This basically boils down to an AND with an additional filter. Compatibility is again checked as above in the AND case.
  - The second case happens if $\theta_1 \in [[P_1]]_{DS}$ and there is no compatible substitution in $[[P_2]]_{DS}$. Then this θ_1 is in $[[P']]_{DS}$. In an auxiliary predicate ($\mathrm{ans'_i}$) we derive all substitutions of $[[P_1]]_{DS}$, for which a compatible substitution exists in $[[P_2]]_{DS}$. For this case we derive the corresponding ASP atom of θ_1 for P' and position i, where all

variables which do occur in $vars_i^{false}(P')$ but not in P_1 are set to null and additionally we exclude those substitutions with compatible substitutions in $[[P_2]]_{DS}$ by the use of default negation with the auxiliary predicate.

– In the third case a substitution θ_1 is in $[[P']]_{DS}$ if $\theta_1 \in [[P_1]]_{DS}$ and there exists a compatible substitution $\theta_2 \in [[P_2]]_{DS}$, s.t. the filter does not hold for $(\theta_1 \cup \theta_2)$. This is directly matched by the third rule in the translation, i.e. we derive the corresponding atom for θ_1 for the pattern P' at position i, if θ_1 has a corresponding atom for P_1 at position $2 * i$ and we can derive a corresponding atom for θ_2 for P_2 at $2 * i + 1$ and these two are compatible and additionally the filter is not satisfied.

- If $P' = (P_1 \text{ UNION } P_2)$, then the translation simply derives each substitution from $[[P_j]]_{DS}$ for $j \in \{1, 2\}$ and replaces variables not occurring in a pattern by null. And in addition we add the integers 1 and 2 for each "branch" of the UNION, s.t. duplicates are preserved. Thus the claim follows.

□

4.2 Translation of Solution Modifiers

In this section we will show how to encode a solution modifier $SM = (\vec{V}, S, dst, l, o)$ on top of the core translation. This translation is more involved due to the necessity of defining orderings of solutions. It is also less declarative than the remaining encodings due to its imperative nature of ordering in the exact manner as defined in the specification (Harris & Seaborne, 2013, Section 15). The translation consists of several modules for clearer readability. We make use of an ASP technique for iterating through all elements in a set via ordering first the elements in this set and then deriving a successor predicate. Some more details of this technique can be found e.g. in (Eiter, Ianni, & Krennwallner, 2009). The first module is used for comparing two substitutions θ and θ' with respect to their ordering. To this end we define the predicate $\texttt{lt}_\texttt{i}(\overline{X}, \overline{X'})$, which orders the substitutions lexicographically accordingly to $\vec{V}$, i.e. X is ordered lower than X'. For this to work, we introduce an ordering rule for comparing the j-th element in $\vec{V} = (v_1, ..., v_j, ..., v_n)$ for each element in the vector. We define $V_{j,n} = \{v_j, ..., v_n\}$. Note that we always assume that $\vec{V}$ covers exactly all $v \in vars_i^{dst}(P)$ and that the auxiliary variables we use in the modules (O and O_i) are distinct from the other variables occurring in patterns.

$$lowerThan(\vec{V}, P, dst, D, i, j) =$$
$$\texttt{lt}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)[V_{j,n} \to V'_{j,n}]}, D) \text{ :- } \texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D), \texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)[V_{j,n} \to V'_{j,n}]}, D), v_j < v'_j.$$

The next ASP module is used for finding duplicates. If $dst = false$ we will set this module to be empty, otherwise we will derive the predicate $\texttt{remove}_\texttt{i}$ to mark each duplicate to be removed. The rule for this purpose is very simple, since we want to remove all duplicates, except for the first one occurring in the ordered list of duplicates. This can be achieved by exploiting just $\texttt{lt}_\texttt{i}$.

A substitution is a duplicate, which should be removed, if it is equal on the projected variables to another one, which is ordered before.

$$
\begin{aligned}
&removeDuplicates(P, S, true, D, i) = \\
&\texttt{remove}_\texttt{i}(\overline{vars_i^{true}(P)}, D) \ \texttt{:-} \\
&\qquad \texttt{lt}_\texttt{i}(\overline{vars_i^{true}(P)}[(vars_i^{true}(P) \setminus S) \rightarrow (vars_i^{true}(P) \setminus S)'], \overline{vars_i^{true}(P)}, D).\ ^{\dagger}
\end{aligned}
$$

The $order$ module now derives the successor and infimum predicates $\texttt{succ}_\texttt{i}$ and $\texttt{inf}_\texttt{i}$ respectively via helper predicates. A substitution θ'' is not a successor of θ if there is a substitution θ' ordered in between those two. Additionally θ' must not be marked for removal. Similarly we derive the infimum.

$$
\begin{aligned}
&order(P, dst, D, i) = \\
&\texttt{notsucc}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)}'', D) \ \texttt{:-} \\
&\qquad \texttt{lt}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)}', D), \\
&\qquad \texttt{lt}_\texttt{i}(\overline{vars_i^{dst}(P)}', \overline{vars_i^{dst}(P)}'', D), \\
&\qquad \texttt{not}\ \texttt{remove}_\texttt{i}(\overline{vars_i^{dst}(P)}', D). \\
&\texttt{succ}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)}', D) \ \texttt{:-} \\
&\qquad \texttt{lt}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)}', D), \\
&\qquad \texttt{not}\ \texttt{notsucc}_\texttt{i}(\overline{vars_i^{dst}(P)}, \overline{vars_i^{dst}(P)}', D), \\
&\qquad \texttt{not}\ \texttt{remove}_\texttt{i}(\overline{vars_i^{dst}(P)}, D), \texttt{not}\ \texttt{remove}_\texttt{i}(\overline{vars_i^{dst}(P)}', D). \\
&\texttt{notinf}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-}\ \texttt{lt}_\texttt{i}(\overline{vars_i^{dst}(P)}', \overline{vars_i^{dst}(P)}, D), \\
&\qquad \texttt{not}\ \texttt{remove}_\texttt{i}(\overline{vars_i^{dst}(P)}', D). \\
&\texttt{inf}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \ \texttt{:-}\ \texttt{not}\ \texttt{notinf}_\texttt{i}(\overline{vars_i^{dst}(P)}, D), \texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D). \\
&\qquad \texttt{not}\ \texttt{remove}_\texttt{i}(\overline{vars_i^{dst}(P)}, D).
\end{aligned}
$$

For computing the slice of LIMIT and OFFSET we need to assign integers to the ordering, beginning with 1. This is straightforward with the help of the infimum and successor predicates.

$$
\begin{aligned}
&assignInteger(P, dst, D, i) = \\
&\texttt{ans}_\texttt{i}^\texttt{o}(\overline{vars_i^{dst}(P)}, D, 1) \ \texttt{:-}\ \texttt{inf}_\texttt{i}(\overline{vars_i^{dst}(P)}, D). \\
&\texttt{ans}_\texttt{i}^\texttt{o}(\overline{vars_i^{dst}(P)}, D, O) \ \texttt{:-}\ \texttt{ans}_\texttt{i}^\texttt{o}(\overline{vars_i^{dst}(P)}', D, O'), \texttt{succ}_\texttt{i}(\overline{vars_i^{dst}(P)}', \overline{vars_i^{dst}(P)}, D), \\
&\qquad\qquad\qquad O = O' + 1.
\end{aligned}
$$

We collect all the necessary modules for solution modifiers in the module $SolutionModifiers$.

[†] Simple duplicate removal by projection must preserve the ordering of the non-projected solutions, since the slice must be applied w.r.t. the original ordering (specification (Harris & Seaborne, 2013, Section 15)). Hence we retain the non-projected variables during the application of solution modifiers and only afterwards apply the projection.

$SolutionModifiers(\vec{V}, S, P, dst, D, i) =$
$assignInteger(P_1, dst, D, i) \cup order(P_1, dst, D, i) \cup removeDuplicates(P_1, S, dst, D, i) \cup$
$\bigcup_{1 \leq j \leq |vars_i^{dst}(P)|} lowerThan(\vec{V}, P_1, dst, D, i, j)$

Let $Q = (DS, P, SM)$ be a query and $DS = (G, G_{named})$ with $SM = (\vec{V}, S, false, l, o)$. We translate this query to a logic program Π_Q by just collecting all necessary modules and adding the "slice" rule, which removes solutions that do not fall under the range. If $l = 0$ or $o = 0$ we omit the respective conditions of the slice.

$\Pi_Q = \Pi_{DS} \cup \tau(P, false, \texttt{default}, 2) \cup \Pi^{DS}_{\mathsf{FILTER}} \cup Join(n) \cup$
$SolutionModifiers(\vec{V}, S, P, false, D, 2) \cup$
$\texttt{ans}_1(\overline{vars_1^{true}(P) \cap S}, O_2, D) \texttt{ :- } \texttt{ans}_2^{\circ}(\overline{vars_2^{false}(P)}, D, O_2), o < O_2, O_2 \leq o + l.$

This logic program computes the query answer in $\texttt{ans}_1$ for n being the maximal cardinality of shared variables in rules in $\tau(P, false, \texttt{default}, 2)$. Note that in the rule for deriving the predicate $\texttt{ans}_1$ we can exclude all auxiliary variables used in other rules and just use O_2 as the identifier for duplicates, hence we can use $vars_1^{true}(P)$ here. The variant for discarding duplicates can now be written as follows:

$\Pi'_Q = \Pi_{DS} \cup \tau(P, true, \texttt{default}, 2) \cup \Pi^{DS}_{\mathsf{FILTER}} \cup Join(n) \cup$
$SolutionModifiers(\vec{V}, S, P, true, D, 2) \cup$
$\texttt{ans}_1(\overline{vars_1^{true}(P) \cap S}, D) \texttt{ :- } \texttt{ans}_2^{\circ}(\overline{vars_2^{true}(P)}, D, O_2), o < O_2, O_2 \leq o + l.$

In Section 4.3 we will use this idea for defining the translation of subquery patterns.

4.3 Translation of SPARQL1.1 Features

In this section we will show how to translate the new features of SPARQL1.1 to ASP. We will start with the property paths, the VALUES and MINUS patterns and then proceed to subqueries and the new FILTER expression, which are more involved in their encoding. The BIND translation, which requires non-pure ASP encodings via external atoms, due to the possibility of introducing new values not present as terms in the query or graph is shown in the next section.

First we need to extend the indexed variable set by introducing auxiliary variables also for the VALUES pattern (I_i) and the sequential (S_i) and alternative property paths (U_i), as well as variable projection to ensure the correct variable scope of subqueries, including another auxiliary variable (O_i).

Definition 4.6 (Extended indexed variable set) *Given a positive integer i, $dst \in \{true, false\}$, a property path PP and a graph pattern P, we extend the indexed variable set $vars_i^{dst}(P)$ as follows:*

- *For $P = (P_1$ MINUS $P_2)$ let $vars_i^{dst}(P) = vars_{2*i}^{dst}(P_1)$.*
- *For $P = (s, PP, o)$, let $vars_i^{dst}(P) = vars(\{s, o\}) \cup vars_{2*i}^{dst}(PP)$.*

- *For* $P = (P_1$ **BIND** $expr$ **AS** $x)$, *let* $vars_i^{dst}(P) = vars_{2*i}^{dst}(P_1) \cup \{x\}$.
- *For* $P = ($**VALUES** $\vec{V}\, D)$, *let* $vars_i^{false}(P) = \{x \mid x \in \vec{V}\} \cup \{I_i\}$.
- *For* $P = ($**VALUES** $\vec{V}\, D)$, *let* $vars_i^{true}(P) = \{x \mid x \in \vec{V}\}$.
- *For* $P = (P_1, (\vec{V}, S, false, l, o))$ *let* $vars_i^{dst}(P) = (S \cap vars_{2*i}^{true}(P_1)) \cup \{O_i\}$.
- *For* $P = (P_1, (\vec{V}, S, true, l, o))$ *let* $vars_i^{dst}(P) = (S \cap vars_{2*i}^{true}(P_1))$.

For subqueries an auxiliary variable (O_i) is introduced to indicate the ordering. We preserve this variable if $dst = false$. This variable is also used as an identifier for duplicates as in Section 4.2, here we just add this mechanism in general for subqueries, which can occur anywhere as a subpattern now.

For property paths auxiliary variables are required for duplicate solutions. Here, similarly as for the UNION pattern, we introduce a new variable for the sequential path, the alternative path, as well as for the negated path containing both normal and inverse IRIs. For the latter, the introduced variable actually corresponds to the auxiliary variable of UNION pattern to which the paths are rewritten. In all other cases no new variables are introduced.

Definition 4.7 (Indexed variable set for property paths) *Given a positive integer* i, $dst \in \{true, false\}$, *a property path* PP, *we define the indexed variable set for property paths* $vars_i^{dst}(PP)$ *as follows:*

- *Let* $vars_i^{true}(PP) = \emptyset$
- *For* $PP = PP_1/PP_2$ *let* $vars_i^{false}(PP) = vars_{2*i}^{false}(PP_1) \cup vars_{2*i+1}^{false}(PP_2) \cup \{S_i\}$
- *For* $PP = PP_1|PP_2$ *let* $vars_i^{false}(PP) = vars_{2*i}^{false}(PP_1) \cup vars_{2*i+1}^{false}(PP_2) \cup \{U_i\}$
- *For* $PP =!(N, N')$ *let* $vars_i^{false}(PP) = vars_{2*i}^{false}(PP_1) \cup vars_{2*i+1}^{false}(PP_2) \cup \{U_i\}$
- *else let* $vars_i^{false}(PP) = \emptyset$

We now can define a translation of property paths. We require an auxiliary predicate `reach`, which we use for recursively applying a property path PP. The variables X_i and Y_i are assumed to be fresh.

(7) Let $P = (s, PP_1/PP_2, o)$ then $\tau(P, dst, D, i) =$
$\tau((s, PP_1, S_i)$ **AND** $(S_i, PP_2, o), dst, D, i)$

(8) Let $P = (s, PP_1|PP_2, o)$ then $\tau(P, dst, D, i) =$
$\tau((s, PP_1, o)$ **UNION** $(s, PP_2, o), dst, D, i)$

(9) Let $P = (s, \hat{}PP, o)$ then $\tau(P, dst, D, i) =$
$\tau((o, PP, s), dst, D, i)$

(10) Let $P = (s, !(N, N'), o)$ then $\tau(P, dst, D, i) =$
$\tau((s, !(N), o)$ **UNION** $(o, !(N'), s), dst, D, i)$

(11) Let $P = (s, !(N), o)$ and $N = \{p_1, ..., p_m\}$ and X_i a fresh variable then $\tau(P, dst, D, i) =$

$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $\texttt{triple}(s, X_i, o, D), X_i \neq p_1, ..., X_i \neq p_m$

For the following translation of variable length paths we can set $dst = true$ for the subpatterns, since we are not interested in duplicate solutions.

(12) Let $P = (s, PP?, o)$ then $\tau(P, dst, D, i) =$
$\tau((s, PP, o), true, D, 2 * i) \cup$

$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $\texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}((s, PP, o))}, D)$
$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $term(s), s = o.$

(13) Let $P = (s, PP+, o)$ then $\tau(P, dst, D, i) =$
$\tau((X_i, PP, Y_i), true, D, 2 * i) \cup$

$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $\texttt{reach}_i(s, o, D)$
$\texttt{reach}_i(X_i, Y_i, D)$:- $\texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}((X_i, PP, Y_i))}, D).$
$\texttt{reach}_i(Z_i, Y_i, D)$:- $\texttt{reach}_i(Z_i, X_i, D), \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}((X_i, PP, Y_i))}, D).$

(14) Let $P = (s, PP^*, o)$ then $\tau(P, dst, D, i) =$
$\tau((X_i, PP, Y_i), true, D, 2 * i) \cup$

$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $\texttt{reach}_i(s, o, D)$
$\texttt{reach}_i(X_i, Y_i, D)$:- $\texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}((X_i, PP, Y_i))}, D).$
$\texttt{reach}_i(Z_i, Y_i, D)$:- $\texttt{reach}_i(Z_i, X_i, D), \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{true}((X_i, PP, Y_i))}, D).$
$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D)$:- $term(s), s = o.$

For the special case that both s and o are not variables, we replace the last added rule of **(12)** and **(14)** by just a fact $\texttt{ans}_\texttt{i}(D)$ if $s = o$.

For the MINUS pattern we can compute in an auxiliary predicate all compatible substitutions and then use default negation to exclude them, similarly as in the difference operator in the relational algebra. According to Definition 2.14 we have one additional aspect to consider, namely we have to check for disjoint domains of the substitutions. In the ASP encoding this means that two ground atoms $\texttt{ans}_{\texttt{2*i}}$ and $\texttt{ans}_{\texttt{2*i+1}}$ represent substitutions with disjoint domains if we can join them as before and additionally every shared term is null in one or both of the atoms. For computing this we utilize a similar encoding as for the $\texttt{join}_n$ predicate:

$Unbound(n) =$

$\texttt{unbound}(X, X, X)$:- $term(X), \texttt{not}\ null(X).$ $\texttt{unbound}(\texttt{null}, \texttt{null}, \texttt{null}).$
$\texttt{unbound}(X, \texttt{null}, \texttt{null})$:- $term(X).$
$\texttt{unbound}(\texttt{null}, X, \texttt{null})$:- $term(X).$
$\texttt{unbound}_1(X_1', X_1'', X_1)$:- $\texttt{unbound}(X_1', X_1'', X_1).$
$\texttt{unbound}_2(X_1', X_2', X_1'', X_2'', X_1, X_2)$:- $\texttt{unbound}_1(X_1', X_1'', X_1), \texttt{unbound}(X_2', X_2'', X_2).$
$\texttt{unbound}_3(X_1', X_2', X_3', X_1'', X_2'', X_3'', X_1, X_2, X_3)$:- $\texttt{unbound}_2(X_1', X_2', X_1'', X_2'', X_1, X_2), \texttt{unbound}(X_3', X_3'', X_3).$
$\vdots$
$\texttt{unbound}_n(X_1', \ldots, X_n', X_1'', \ldots, X_n'', X_1, \ldots, X_n)$:-
$\quad \texttt{unbound}_{n-1}(X_1', \ldots, X_{n-1}', X_1'', \ldots, X_{n-1}'', X_1, \ldots, X_{n-1}), \texttt{unbound}(X_n', X_n'', X_n).$

That is we can compute if a variable is bound in one of the substitutions. We will write $\mathsf{null}_n = \underbrace{\mathsf{null}, ..., \mathsf{null}}_{n \text{ times}}$ shorthand for n many null terms. Now we can define the translation as follows:

(15) Let $P = (P_1 \ \mathsf{MINUS} \ P_2)$ then $\tau(P, dst, D, i) \ =$
$\tau(P_1, D, dst, 2 * i) \ \cup \ \tau(P_2, dst, D, 2 * i + 1) \cup$

$\mathtt{ans_i}(\overline{vars_i^{dst}(P)}, D) \ \mathtt{:-} \ \mathtt{ans_{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D), \ \mathtt{not} \ \mathtt{ans_i}'(\overline{vars_i^{true}(P_1)}, D).$
$\mathtt{ans_i}'(\overline{vars_i^{true}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D) \ \mathtt{:-}$
$\quad \mathtt{ans_{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D),$
$\quad \mathtt{ans_{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D),$
$\quad \mathtt{not} \ \mathtt{ans_i}''(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], \overline{vars_{2*i+1}^{dst}(P_1)}[S_{P_1,P_2} \to S''_{P_1,P_2}], \mathsf{null}_{|S_{P_1,P_2}|}, D),$
$\quad \mathtt{join}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}).$
$\mathtt{ans_i}''(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], \overline{vars_{2*i+1}^{dst}(P_1)}[S_{P_1,P_2} \to S''_{P_1,P_2}], \overline{S_{P_1,P_2}}^u, D) \ \mathtt{:-}$
$\quad \mathtt{ans_{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}[S_{P_1,P_2} \to S'_{P_1,P_2}], D),$
$\quad \mathtt{ans_{2*i+1}}(\overline{vars_{2*i+1}^{dst}(P_2)}[S_{P_1,P_2} \to S''_{P_1,P_2}], D),$
$\quad \mathtt{join}_{|S_{P_1,P_2}|}(\overline{S'_{P_1,P_2}}, \overline{S''_{P_1,P_2}}, \overline{S_{P_1,P_2}}), \mathtt{unbound}_{|S_{P_1,P_2}|}(\overline{S_{P_1,P_2}}', \overline{S_{P_1,P_2}}'', \overline{S_{P_1,P_2}}^u).$

That is we compute in $\mathtt{ans_i}''$ for pairs of substitutions a sequence $\overline{S_{P_1,P_2}}^u$, which is null if one of the substitutions is undefined via null, too.

The VALUES pattern can be easily translated by just adding facts for each substitution, but add an identifier I_i to the variables to distinguish between duplicates given in the table Ds. We set this identifier to be an index from 1 to the number of tuples in Ds.

(16) Let $P = (\mathsf{VALUES} \ \vec{V} Ds)$ then $\tau(P, false, D, i) \ =$
$\{\mathtt{ans_i}(\overline{vars_i^{false}(P)}[x \to x\theta, x \in \vec{V}, I_i \to j], D). \mid \vec{V}\theta \in Ds, j \text{ is the index of } \theta \text{ in } Ds\}$

Let $P = (\mathsf{VALUES} \ \vec{V} Ds)$ then $\tau(P, true, D, i) \ =$
$\{\mathtt{ans_i}(\overline{vars_i^{true}(P)}[x \to x\theta, x \in \vec{V}], D). \mid \vec{V}\theta \in Ds\}$

A subquery is now derived very similarly as in Section 4.2 by just collecting all necessary modules for the solution modifiers and adding the "slice" rule, which removes solutions that do not fall under the range. Note that we implicitly projected variables here via the indexed variable set $vars_i^{dst}$. We preserve the order induced by O_i if $dst = false$ and thus have an identifier for each duplicate. If $l = 0$ or $o = 0$ we omit the respective conditions of the slice.

(17) Let $P = (P_1, SM)$ a subquery with $SM = (\vec{V}, S, dst, l, o)$ then $\tau(P, dst', D, i) \ =$
$SolutionModifiers(\vec{V}, S, P_1, dst, D, 2 * i) \cup \tau(P_1, dst, D, 2 * i) \cup$

$\mathtt{ans_i}(\overline{vars_i^{dst}(P)}, D) \ \mathtt{:-} \ \mathtt{ans^o_{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D, O_i), o < O_i, O_i \leq o + l.$

Finally for this section, we will present the translation of the new atomic filter expression (EXISTS P). Here a substitution θ satisfies the filter if $[[P\theta]]_{DS}$ is non-empty. The complex task is to translate the substitution applied to a pattern as defined in the semantics, i.e. $P\theta$. The basic

idea is to parameterize the computation of P by θ, this means adding variables representing θ to every predicate in a rule for the computation of P. Further we add to the body of those rules, where a substitution of P by θ can occur, the predicate representing θ itself to bind the variables. We also need joins for checking compatibility of θ with the current solution. Note that $P\theta$ has some special cases, where a substitution does not take place, e.g. variables in a BIND or VALUES pattern are not substituted, since this would lead to a syntactically wrong pattern. That is patterns that potentially change by the substitution are triple patterns, negated property paths, atomic filter expressions, expressions in BIND patterns and the graph variable in a GRAPH pattern. For more complex patterns such as AND we propagate the applied substitution of the subpatterns. Thus we specify for which θ the current pattern is computed. This is necessary because variables in P might otherwise be unbound, e.g. variables occurring only in an atomic filter inside P.

We define an auxiliary function for applying the substitution to rules of our translation: Let $r = H(r)$:- $B(r)$ be an ASP rule with a non-empty body, i and j be integers, P a graph pattern and $join \in \{true, false\}$, then

$$Substitute(r, i, j, P, join) = H'(r) \text{ :- } B'(r)$$

Here, $H'(r)$ and $B'(r)$ are defined as follows: for $p_m \in \{\texttt{ans}_\texttt{m}, \texttt{ans}'_\texttt{m}, \texttt{filter}_\texttt{m}, \texttt{reach}_\texttt{m}, \texttt{notinf}_\texttt{m}, \texttt{inf}_\texttt{m}, \texttt{ans}^\circ_\texttt{m}, \texttt{lt}_\texttt{m}, \texttt{ans}''_\texttt{m}, \texttt{notsucc}_\texttt{m}, \texttt{succ}_\texttt{m}\}$ and m is an integer for which it holds that $m > i$ then:

- If the head is of the form $H(r) = p_m(t_1, \ldots, t_k)$ then $H'(r) = p_{m,j}(t_1, \ldots, t_k, \overline{vars_i^{true}(P)}^{i,j})$
- If $p_m(t_1, \ldots, t_k) \in B(r)$, then rewrite it to $p_{m,j}(t_1, \ldots, t_k, \overline{vars_i^{true}(P)}^{i,j})$
- $B'(r) = B(r) \cup \{\texttt{ans}_\texttt{i}(\overline{vars_i^{true}(P)}^{i,j}, D)\}$
- if $join = true$ and V is the set of variables in the head, let $Y \subseteq V$ be the variables in the head, which were renamed with a previous application of $Substitute$ and $X = (V \cap vars_i^{true}(P)) \setminus Y$ then we additionally add to the body the join predicate: $\texttt{join}_{|\texttt{X}|}(X, X^{i,j}, X^{i,j,*})$.

Let $P = (P_1 \text{ FILTER } R)$ be a graph pattern at position i, where a filter expression $R' = (\text{EXISTS}(P_2))$ occurs at index j of the parse tree of R, i.e. R' is a sub filter expression of R, then $\tau(P, dst, D, i)$ is changed by the following steps, where conflicts of several EXISTS are resolved by applying them in order in which they occur in the parse tree in a top-down manner, i.e. applying those with a lower index first.

1. We add the variables from P_1 to the rules evaluating the filter expressions, which contain R', i.e. $vars(R'')$ is extended by $vars_{2*i}^{true}(P_1)$ if R'' contains R'.
2. $Cond(R', v, i, j) = filter_{i,j}(\overline{vars_{2*i}^{true}(P_1)}, v)$.

3. We add the following rules to $\tau(P, dst, D, i)$ to express the value of the filter expression for each of the substitutions of P_1.

$$\texttt{filter}_{\texttt{i,j}}(\overline{vars^{true}_{2*i}(P_1)}', true) \text{ :- } \texttt{ans}_{\texttt{2*i+1,j}}(\overline{vars^{true}_{2*i+1}(P_2)}, D, \overline{vars^{true}_{2*i}(P_1)}').$$
$$\texttt{filter}_{\texttt{i,j}}(\overline{vars^{true}_{2*i}(P_1)}, false) \text{ :- } \texttt{ans}_{\texttt{2*i}}(vars^{true}_{2*i}(P_1), D), \texttt{not filter}_{\texttt{i,j}}(\overline{vars^{true}_{2*i}(P_1)}, true).$$

Thus, the predicate $\texttt{ans}_{\texttt{2*i+1,j}}$ represents the computation of P_2, parameterized by substitutions of P_1, which will be introduced next.

4. We add a modified $\tau(P_2, true, D, 2 * i + 1)$, where we change every rule in $r \in \tau(P_2, true, D, 2 * i + 1)$ as follows:

   (a) If r was introduced by **(1)**, **(11)**, then we modify it to $Substitute(r, 2 * i, j, P_1, true)$.

   (b) If r was introduced by **(12)** and **(14)** and was before any modifications of EXISTS were applied of the form $\texttt{ans}_{\texttt{i'}}(\overline{vars^{dst}_{i'}(P_3)}, D)$:- $term(s), s = o.$, then we modify it to $Substitute(r, 2 * i, j, P_1, true)$. For the special case of both s and o not being variables, we omit the atom $term(s)$, the join and the equation $s = o$ of the rule.

   (c) Each fact $r = \texttt{ans}_{\texttt{i'}}(\overline{V}, D)$ added by a VALUES pattern evaluation is changed to $\texttt{ans}_{\texttt{2*i'}}(\overline{V}, D)$. This change is done only once, even if multiple EXISTS are present in subpatterns. Further the first modification also adds the rule

   $$\texttt{ans}_{\texttt{i',j}}(\overline{V}, D, \overline{vars^{true}_{2*i}(P_1)}^{i,j}) \text{ :- } \texttt{ans}_{\texttt{2*i'}}(\overline{V}, D), \texttt{ans}_{\texttt{2*i}}(\overline{vars^{true}_{2*i}(P_1)}^{i,j}, D).$$

   Subsequent modifications by EXISTS are not allowed to change the atom $\texttt{ans}_{\texttt{2*i'}}(\overline{V}, D)$.

   (d) If r was introduced by **(5)**, then we modify it to $Substitute(r, 2 * i, j, P_1, true)$, but set the shared variables to be the graph variable g. If g is not a variable, then this step can be omitted.

   (e) The recursive definition of filters is changed by two steps. First we do not replace variables with null if they occur in $vars^{true}_{2*i}(P_1)$ in the translations **(5)** and **(6)**. Second we change for an atomic filter R'' the condition $Cond(R'', v, i', j')$ to $Cond'(R'', v, i', j') = Cond(R'', v, i', j') \cup \{\texttt{ans}_{\texttt{2*i}}(\overline{vars^{true}_{2*i}(P_1)}, D)\}$. In this case, we also have to set these not renamed variables to be equal to the renamed ones in the head and other predicates in the rule where $Cond(R'', v, i', j')$ appears. This can be done by multiple applications of '='.

   Subsequent modifications by other EXISTS behave slightly different, namely the variables X in an atom added in this step are renamed to $X^{i,j}$ for X being the variables shared with a previously added atom for this atomic filter and another EXISTS, i.e. X does not influence the atomic filter, since the variables already have been applied.

   (f) Else we replace r by $Substitute(r, 2 * i, j, P_1, false)$.

Example 4.2 *The complex modification for EXISTS expressions is exemplified in Example 2.4 and its translation in the appendix. In particular consider the following subpattern of the EXISTS expression,*

```
(?P foaf:knows ?F)
```

which is dealt with in the translation by the rule with $\texttt{ans}_{6,1}$ *in the head. The unmodified rule would be*

$$\texttt{ans}_6(F, P, default) \ \texttt{:-}\ \texttt{triple}(P, \texttt{foaf:knows}, F, default).$$

which is modified to

$$\begin{aligned}\texttt{ans}_{6,1}(F, P, M21, P21, default) \ \texttt{:-}\ & \texttt{triple}(P, \texttt{foaf:knows}, F, default),\\ & \texttt{ans}_2(M21, P21, default),\\ & \texttt{join}_1(P, P21, P'').\end{aligned}$$

Note that now the predicate in the head incorporates 1 *from the filter for unique naming and also the variables from the pattern preceding the EXISTS expression.*

Finally, Let $Q = (DS, P, SM)$, with $SM = (\vec{V}, S, dst, l, o)$ be a query and $DS = (G, G_{named})$ as defined above. We translate this query to a logic program Π_Q defined as follows. Let $Q' = (P, SM)$.

$$\Pi_Q = \Pi_{DS} \cup \tau(Q', dst, \texttt{default}, 1) \ \cup\ \Pi^{DS}_{\mathsf{FILTER}} \cup Join(n) \cup Unbound(n)$$

This logic program computes the query answer in $\texttt{ans}_1$ for n being the maximal cardinality of shared variables in rules in $\tau(Q, dst, \texttt{default}, 1)$. As before, to achieve a set-semantics instead of multi-set semantics, one can just set the dst flag to be always true. In general one may observe that duplicate solutions can only arise from (a) UNION patterns, (b) projections or (c) duplicates in a VALUES table.

4.4 Translation of BIND

The BIND pattern is different to all the other patterns since it may introduce values not present in the original query and dataset. Thus pure ASP cannot cope with this behavior. For the following translation to work, we base the evaluation of the assignment on external predicates. The main external predicate is

$$\texttt{eval}[expr, x_1, ..., x_n](v)$$

where $expr$ is an assignment expression enclosed in quotes and $v, x_1, ..., x_n$ ASP constants. The atom is true if v is the value of the expression $expr\theta$ evaluated under the substitution $\theta = [?1 \rightarrow x_1, ..., ?n \rightarrow x_n]$, where $?j$ for an integer j are (lexically ordered by variable name) variables occurring in the expression $expr$. For instance

$$\texttt{eval}[\text{"?1+?2"}, 2, 3](v)$$

is true if $v = 5$. In this way we can straightforwardly encode the BIND pattern itself as

(18) Let $P = (P_1 \text{ BIND } expr \text{ AS } x)$ then $\tau(P, dst, D, i) = \tau(P_1, dst, D, 2 * i) \cup$

$$\texttt{ans}_\texttt{i}(\overline{vars_i^{dst}(P)}, D) \texttt{ :- } \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D), \texttt{eval}[expr', \overline{vars_{2*i}^{dst}(P_1)}](v).$$

and $expr'$ is the expression $expr$ where we replace every $v \in Var$ with $?j$ for j the position in the lexicographic ordering of $\overline{vars_{2*i}^{dst}(P_1)}$. Note that error handling applies to this external predicate, i.e. if the expression returns an error then it is only true if $v = \text{null}$.

The introduction of BIND has implications over the whole translation τ, which must be extended as follows; we will refer to the extended translation as τ'. The predicates $\texttt{Join}_n$, $\texttt{Unbound}_n$, equals, lowerThan, isIRI, IsLiteral, isBlank, bound and the binary comparison operators '$=$', '$\neq$' and '$<$' now have to be external predicates as well, since the regular ASP predicates cannot cope with new values produced by BIND. That is we replace $\texttt{equals}(X, Y, v)$ by $\texttt{equals}[X, Y](v)$, which returns $v \in \{true, false, err\}$ in the same manner as the regular ASP predicate is defined, except that an external function decides the value also for new values created by BIND. Similarly we proceed for the other predicates. Note that, since we already use external predicates, we can make use of the external atom rdf for retrieving RDF triples.

Furthermore some special care is needed for FILTER expressions. if R is an atomic filter (6.1 in Section 4), then we change the condition $Cond$ to $Cond'(R, v, i, j) = \{Cond(R, v, i, j), \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P')}, D)\}$ if P' is the graph pattern at position $2 * i$ of the query parse tree and $x \in vars(R)$ is an assigned variable by the application of $(P'' \text{ BIND } expr \text{ AS } x)$, a subpattern of P'. That is, we add the predicate $\texttt{ans}_{\texttt{2*i}}$ to the body containing the atomic filter evaluation. This ensures that also new values are considered in the filter.

The BIND pattern is also a special case for the modifications necessary by the EXISTS filter expression from Section 4.3. Let $(P_2 \text{ FILTER } R)$ be a filter pattern at position i' and $R' = (\text{EXISTS } P_3)$ be a subfilter expression of R at position j. If $P = (P_1 \text{ BIND } expr \text{ AS } x)$ is a subpattern of P_3 at position i then a rule r introduced by **(18)** is changed to $Substitute(r, i', j, P_2, true)$. Additionally we change the external atom to $\texttt{eval}[expr', \overline{vars_{2*i}^{dst}(P_1) \cup vars_{i'}^{true}(P_2)}[S_{P_{2*i}, P_{i'}} \rightarrow S^{i,j,*}_{P_{2*i}, P_{i'}}]](v)$. We change $expr'$ accordingly to incorporate the new variables. This rule ensures that for the expression in the assignment we also may use the variables from the outer pattern of the EXISTS filter.

Example 4.3 *As an example consider a query containing the patterns* $P_1 = (P_2 \text{ BIND } (?X+?Y) \text{ AS } ?Z)$ *and* $P = (P_1 \text{ FILTER } \neg(5 = ?Z))$ *the latter at position* i*, then the following rules are relevant for exemplifying the changes needed for* BIND*. Assume that* $vars_{4*i}^{dst}(P_2) = \{X, Y\}$.

$$\texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D) \texttt{ :- } \texttt{ans}_{\texttt{4*i}}(\overline{vars_{4*i}^{dst}(P_2)}, D), \texttt{eval}[\text{"?1+?2"}, \overline{vars_{4*i}^{dst}(P_2)}](Z).$$
$$\texttt{ans}_\texttt{i}(vars_i^{dst}(P), D) \texttt{ :- } \texttt{ans}_{\texttt{2*i}}(vars_{2*i}^{dst}(P_1), D), \texttt{filter}_{\texttt{i,1}}(\overline{\{Z\}}, true).$$
$$\texttt{filter}_{\texttt{i,1}}(\overline{\{Z\}}, V) \texttt{ :- } \texttt{equals}[5, Z](V_1), \texttt{ans}_{\texttt{2*i}}(\overline{vars_{2*i}^{dst}(P_1)}, D), neg(V_1, V).$$

Note that we added to the body of the third rule the corresponding predicate $\texttt{ansi}_{2*\texttt{i}}$.

4.5 Query Forms

SPARQL queries are not restricted to SELECT queries, but also allow other result forms, indicated by the keywords ASK and CONSTRUCT.[19] We will discuss these forms, which can be translated to ASP.

4.5.1 SELECT

In our translation we have shown how to translate a query with solution modifiers to ASP. A notable divergence is the direct translation of the SELECT statement itself, namely the projection. As long as we have to preserve duplicates we cannot make a projection in ASP in the way that the variables projected out are also projected in the ASP atoms, since then we would lose the duplicate solutions. In our core translation this means that a projection is only applied if we are faced with a DISTINCT query. Note that here also the DISTINCT keyword itself is applied from "outside" of the parse tree of the pattern. In the extended version with solution modifiers (Section 4.2) we can use the ordering to distinguish duplicates in one single variable, but still auxiliary variables are required.

4.5.2 ASK

Apart from SELECT queries SPARQL offers boolean queries $Q_{\mathsf{ASK}} = (P_{\mathsf{ASK}}, DS_{\mathsf{ASK}}, \emptyset)$ with an solution modifier (denoted by the keyword ASK). We can cater for these in our translation by just adding a new predicate $\texttt{ans}_0$ with arity 0. Thus we add a rule and set $\Pi_{Q_{ASK}} = \Pi_Q \cup \{\texttt{ans}_0 \texttt{ :- } \texttt{ans}_1(\overline{vars_1(P_{\mathsf{ASK}})}, DS_{\mathsf{ASK}}).\}$, with $Q = (P_{ASK}, DS_{ASK}, ((), \emptyset, true, 0, 0))$.

4.5.3 CONSTRUCT

The CONSTRUCT result form, which allows to construct new triples can be emulated in our approach as well. Namely, we can allow queries of the form

$$Q_{\mathsf{C}} = (\mathsf{CONSTRUCT} P_{\mathsf{C}}, P, DS)$$

where P_{C} is a graph pattern (called *CONSTRUCT template*) consisting only of triple patterns. Note that SPARQL allows blank nodes in P_{C} with the implied semantics, that the result graph is obtained by applying each substitution θ in the evaluation of $[[P]]_G$ to each triple in P_{C}. For each blank node $\_:\texttt{x}$ in P_{C}, $\_:\texttt{x}\theta$ is defined as a unique new blank node per substitution. This behavior is illustrated best with an example.

[19] We leave out DESCRIBE, cf. (Harris & Seaborne, 2013, Section 16.4), which is only informative in the specification and does not have a specified semantics.

Example 4.4 *Let us consider the following* CONSTRUCT *query upon* $DS = (\{\texttt{alice.org}\}, \emptyset)$ *which extracts Triples about persons and their names from the source graph and "anonymizes" all subjects by replacing them with blank nodes.*

```
CONSTRUCT  _:b a foaf:Person; foaf:name ?N
FROM <alice.org>
WHERE {  ?X a foaf:Person. ?X foaf:name ?N }
```

The result graph for this query would be (modulo blank node identifiers, which may differ from implementation to implementation):

```
_:b1 a foaf:Person; foaf:name "Alice".
_:b2 a foaf:Person; foaf:name "Bob".
```

For blank nodes occurring in CONSTRUCT templates, we can use a "trick" using Skolemization, i.e. treating the blank node identifier as a Skolem function and constructing a Skolem term from this function name and the solution tuples of P as arguments. However, as in a function-free language such as ASP, this trick does not work out-of-the box. Here is where again the power of the external predicates in ASP_{ex} is needed. Particularly, we will make use of the external predicate $\texttt{sk}[id, v_1, \ldots, v_n](sk_{n+1})$, which we introduce now. We will use this predicate to emulate Skolemization, i.e.

- we replace each blanknode $\_:\texttt{b}$ among s, p, o in rule 2 by a fresh variable X_b, and
- add $\texttt{sk}[\_:\texttt{x}, \overline{vars P}](X_b)$ to the rule body.

The external predicate $\texttt{sk}[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$ from its input parameters. We can use for this string concatenation. That is, the value for sk_{n+1} is then just the string $'id(v_1, \ldots, v_n)'$. In addition we need an external predicate for verifying that a triple is indeed a valid triple as defined in (Harris & Seaborne, 2013, Section 18.1), i.e. the subject and object must be either IRIs, literals or blank nodes and the predicate an IRI. Since blank nodes are introduced via external predicates, this verifying predicate must be external as well to cope with fresh values. Thus we define an external predicate $\texttt{validTriple}[s, p, o](v)$, which is true if (s, p, o) is a valid triple and v is equal to $true$.

We then can simply add a rule:

$$\texttt{triple}(s, p, o, \mathsf{C}) \ \texttt{:-}\ \texttt{ans}_1(\overline{vars(P)}, \texttt{default}), \texttt{validTriple}[s, p, o](true). \tag{2}$$

to Π_Q for each triple pattern (s, p, o) in P_{C}. Note that C is used as a special constant here, indicating a constructed statement. The result graph is then naturally represented in the answer set of the program extended that way in the extension of the predicate $\texttt{triple}(\cdot, \cdot, \cdot, \mathsf{C})$. Note that this rule also guarantees that only valid RDF triples are constructed.

Example 4.5 *For the query from Example 4.4 above we would obtain then the following translation along with* $\Pi^{DS}_{\mathsf{FILTER}}$*:*

```
triple(S,P,O,default) :- rdf["alice"](S,P,O).
ans1(N,X,default) :- ans2(X,default), ans3(N,X,default),.
ans2(X,default) :- triple(X,rdf:type,foaf:Person).
ans3(N,X,default) :- triple(X,foaf:Name,N).
triple(Xb,rdf:type,foaf:Person,C) :- ans1(N,X,default), sk[b,N,X](Xb),
                                     validTriple[Xb,rdf:type,foaf:Person](true).
triple(Xb,foaf:name,N,C) :- ans1(N,X,default), sk[b,N,X](Xb),
                            validTriple[Xb,foaf:name,N](true).
```

The answer set of this program, contains

```
{ triple('b("Alice",alice:me)', rdf:type,foaf:Person,C),
  triple('b("Alice",alice:me)',foaf:name,"Alice",C),
  triple('b("Bob",_:c)',rdf:type,foaf:Person,C),
  triple('b("Bob",_:c)',foaf:name,foaf:Person,C) }
```

where from the result graph above can be extracted by simple post-processing that replaces each constructed term `'b(...)'` *by a unique blank node.*

The resulting programs from the translations of CONSTRUCT queries remain essentially non-recursive, i.e. locally stratified, such that despite the use of external predicates, strong safety (and thus termination/finiteness), as mentioned in the end of Section 3 above, is still guaranteed.

5 Discussion

In this section we will discuss aspects relating to expressiveness of various features of SPARQL in relation to the presented translation, as well as other aspects with regards to related and future work. We will start in Section 5.1 with a summary, relating the different features of SPARQL to different fragments of ASP regarding our translation; this is also related to the discussion of so-called well-designed graph patterns, defined originally in (Pérez et al., 2009). In Section 5.2 we will discuss how our approach can be used to provide a semantics to so-called "extended RDF Graphs", which allow to use SPARQL CONSTRUCT queries themselves as a "rules" within an RDF graph. We will discuss related works with "neighbour" W3C standards RDFS, OWL and RIF in Section 5.3. Finally, implementations of our approach on top of ASP engines but also on top of traditional relational databases is discussed in Section 5.4.

5.1 Relating fragments of SPARQL to fragments of ASP

Table 1 illustrates which SPARQL features require which ASP features in our translation, according to the classes of programs defined in Section 3:

As for columns 1–3 of Table 1, we note that for the core translation which we gave in Section 4 we did not need recursion. In fact, all but OPT graph patterns could be encoded in positive

programs. As for FILTERs, which are not mentioned in the table, we note the we used default negation in the encoding the "=" (equals) and "<" (lowerThan) FILTER functions (cf. program $\Pi^{DS}_{\mathsf{FILTER}}$ in Section 4.1); strictly speaking, only a subset of complete set of FILTER functions in the SPARQL1.1 specification (Harris & Seaborne, 2013, Section 17.3) can be implemented without either external predicates, default negation, or bespoke built-ins in the ASP solver.

The DISTINCT solution modifier essentially switches "on" the set-based semantics, as it has been used in earlier translations from SPARQL to Datalog (Polleres, 2007; Angles & Gutierrez, 2008). In fact, building up on the results of (Angles & Gutierrez, 2008), our translation shows that the results shown for graph pattern matching in SPARQL1.0 under a set based semantics carry over to the bag semantics used in the official W3C specification. For all these features, we would take other semantics such as the well-founded (Gelder, Ross, & Schlipf, 1988) semantics or perfect model semantics (Przymusinski, 1988) into account, which coincides with the answer set semantics on non-recursive programs. Further, as documented in column 4 of Table 1, we note that we needed to use recursion to encode ordering, which is relevant for projection (π, implicit in the SELECT clause) under multiset-semantics and in combination with other solution modifiers (ORDER BY, LIMIT, OFFSET); An exception here is DISTINCT, since this solution modifier essentially enables set-bases semantics, DISTINCT queries not using the other solution modifiers could be encoded without the additional machinery added for ordering and slicing. We note here that the recursion used for ordering is in principle (locally) stratified, such that even here the full power of the answer set semantics would not be needed, strictly speaking.

As for SPARQL1.1 features, VALUES and MINUS are encodable in positive programs or, respectively, programs with (nonrecursive, thus) stratified negation again. While the encoding of property paths is recursive, in fact, property paths need only a very restricted form of recursion, expressible in linear Datalog (Abiteboul, Hull, & Vianu, 1995), which for instance is also available in SQL, since SQL-99 (SQL-99, 1999). BIND patterns need external predicates to evaluate built-in expressions that potentially generate new values not occurring in query dataset.

Subqueries (including subqueries within EXISTS in FILTERs) are not mentioned separately in Table 1; their translation depends essentially on which features (patterns, solution modifiers, etc.) are used in the resp. subquery.

Finally, as for CONSTRUCT result forms (cf. Section 4.5.3) we also used an external predicate to emulate implicit generation of new blank nodes.

5.1.1 Non-well-designed patterns

In the context of fragments SPARQL and ASP, so called *well-designed* patterns also deserve a mention, defining a fragment of SPARQL1.0 which restricts the usage of OPT patterns: Pérez et al. (Pérez et al., 2009) define such "well-behaving" optional patterns as follows:

Definition 5.1 (Well-designed pattern) *A graph pattern P is* well-designed *if the following two conditions hold:*

1. *For every occurrence of a sub-pattern $P' = (P_1 \ \mathsf{OPT} \ P_2)$ of P and for every variable v occurring in P, the following condition holds: if v occurs both in P_2 and outside P' then it also occurs in P_1.*

SPARQL1.0				SPARQL1.1			
Patterns		Solution Modifiers		Patterns			
AND UNION GRAPH	OPT	DISTINCT	ORDER BY LIMIT OFFSET π (SELECT)	VALUES	MINUS	Property Paths	BIND
ASP^{nr}_{pos}	ASP^{nr}	ASP^{nr}	ASP^{lstrat}	ASP^{nr}_{pos}	ASP^{nr}	ASP_{pos}	ASP_{ex}

Table 1: Which SPARQL1.0 and SPARQL1.1 are expressible in which ASP classes (according to our translation)?

2. *For every occurrence of a sub-pattern $P' = (P_1$ UNION $P_2)$ of P and for every variable v occurring in P', the following condition holds: if v occurs outside P' then it occurs in both P_1 and P_2.*

Particularly interesting with regards to our translation is the fact that well-designed patterns intuitively do not require joins over variables, which means that for well-designed patterns, our translation could probably be partially be simplified or optimized, which we leave to future work. In fact, non-well-designed patterns might actually be regarded irrelevant for many cases of common queries, as argued in (Pérez et al., 2009).

However, also in relation to SPARQL1.1, it is noteworthy that non-well designed patterns can actually be used to introduce non-monotonic features such as the set difference operator in the language, without using SPARQL1.1's new MINUS or EXISTS features: a well-known way to "emulate" set difference (i.e., "NOT EXISTS" queries in SQL) in SPARQL, which is also pointed out in the official specification, is by using a combination of OPT patterns with a negated BOUND in a FILTER expression.

We illustrate this by the following query which looks for all `foaf:Persons` **without** a nickname in a an RDF graph.

Example 5.1 *As stated in (Prud'hommeaux & Seaborne, 2008, Section 11.4.1), this query over dataset $DS = (\{$`alice.org`$\}, \emptyset)$ can be formulated as follows.*

```
SELECT ?X  FROM <alice.org>
WHERE {  ?X a foaf:Person. OPT { ?X foaf:nick ?N }
         FILTER ( ! BOUND ( ?N ) ) }
```

Obviously, this query is not well-designed in the above sense. Note that the same non-monotonic effect can be achieved even without FILTERs but by using another non-well-designed pattern.

Example 5.2 *Let us assume the named graph `boundchecker.org` be given by the single triple* (_:x, `:is`, `:unbound`).[20]

[20] Actually, the predicate and object constants in that graph do not play a role, it is only important that the subject is a blank node.

```
SELECT ?X
FROM <alice.org>
FROM NAMED <boundchecker>
WHERE { ?X a foaf:Person. OPT { ?X foaf:nick ?N }
        GRAPH <boundchecker> ?N :is :unbound }
```

Now observe, that blank node in Graph `boundchecker.org` *only joins only to unbound variables from the OPT pattern, simply because blank nodes from different graphs (the default graph and the named graph), can never join.*

Interestingly, the latter example contradicts some early conjectures made before the official formal definition of the SPARQL semantics was fixed: Proposition 1 in (Schenk & Staab, 2007) stated that non-monotonicity in SPARQL can exclusively arise from the use of BOUND in filters in combination with OPT, which was later revised and adapted in (Schenk & Staab, 2008), where this proposition is removed.

The discussion around well-designed patterns as a well-behaving fragment of SPARQL is still ongoing, see for instance (Letelier, Pérez, Pichler, & Skritek, 2012). However, we note that probably the definitions and considerations on well-designedness need to be extended on new features within SPARQL1.1, such as EXISTS, MINUS, and also BIND, which might be in a similar sense non-compositional as OPT. We leave this to future work.

5.2 SPARQL as a Rules Language

As stated in earlier works (Schenk & Staab, 2008; Polleres et al., 2007) SPARQL itself may be viewed as an expressive rules language on top of RDF: CONSTRUCT statements have an obvious similarity with view definitions in SQL, and thus may be seen as rules themselves.

Intuitively, in the translation of CONSTRUCT we "stored" the new triples in a new triple outside the dataset DS, defining a new "context" C. We can imagine a similar construction in order to define the semantics of queries over datasets mixing such CONSTRUCT statements with RDF data in the same RDF file.

An example of such an "extended RDF Graph", shown in Figure 2, would be a FOAF file that imports information from another graph, for instance, the following extended graph, would enrich the author's FOAF information by adding implicit foaf:knows relations to co-authors according to DBLP.[21]

Let us assume such a mixed file containing CONSTRUCT rules and RDF triples web-accessible at IRI g, and a query $Q = (DS, P, SM)$, with $DS = (G, G_{named})$. The semantics of a query over a dataset containing g may then be defined by recursively adding Π_{Q_C} to Π_Q for any CONSTRUCT query Q_C in g plus the rules (2) above where the rule head is changed from $\texttt{triple}(s, p, o, \mathsf{C})$ to $\texttt{triple}(s, p, o, g)$. We further need to add a rule

$$\texttt{triple}(s, p, o, default) \texttt{ :- } \texttt{triple}(s, p, o, g).$$

[21] using DBLP's RDF export available at `http://dblp.l3s.de/d2r/`.

```
:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT { :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://dblp.l3s.de/d2r/>
WHERE { ?D dc:creator [ foaf:name ?N ]; }
         dc:creator <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>.
         FILTER (?N != "Axel Polleres".)  }
:me foaf:knows [foaf:name "Herbert Polleres"].
:me foaf:knows [foaf:name "Mechthild Polleres"].
```

Figure 2: Extended RDF Graph containing explicit and implicit triples in Turtle+SPARQL syntax.

for each $g \in G$, in order not to omit any of the implicit triples defined by such "CONSTRUCT rules".

Naturally, the resulting programs possibly involve recursion, and, even worse, recursion over default negation. Fortunately, the general answer set semantics, which we use, can cope with this. We note though, that CONSTRUCT queries with blank nodes such as the one in Example 4.4 are problematic in that case, recursion over the constructed blank nodes becomes possible. Thus, strong safety is no longer guaranteed. Similar problems would arise if CONSTRUCT queries involved BIND patterns. For a more in-depth discussion of extended graphs we refer to (Polleres et al., 2007), where the concept of "extended graphs" was introduced based on our original translation from (Polleres, 2007). Schenk and Staab (Schenk & Staab, 2007, 2008) define and have implemented "networked graphs" – similar to the notion of "extended graphs". Their approach works on top of a standard SPARQL engine on top of which they implement the well-founded semantics, rather than relying on the answer set semantics.

5.3 Entailment Regimes and the Interplay with RDFS, OWL and RIF

The current SPARQL query language specification considers mainly RDF simple entailment (Hayes, 2004, Section 2). Defining "higher" entailment regimes such as RDF entailment is not trivial, since even ordinary RDF entailment (Hayes, 2004, Section 3) is already problematic as a basis for SPARQL query evaluation due to the presence of an infinite set of axiomatic triples; a simple query pattern like $P = (?X, \mathsf{rdf{:}type}, \mathsf{rdf{:}Property})$ would have infinitely many solutions even on the empty (sic!) dataset by matching the infinitely many axiomatic triples in the RDF(S) semantics. The new SPARQL1.1 Entailment Regimes (Glimm et al., 2013) document thus provides reasonable restrictions, that restrict the answers under "higher" entailment regimes, not only for RDF and RDFS, but also for OWL and RIF.

Even more complex issues arise when combining a non-monotonic query language like SPARQL with monotonic ontology languages such as OWL. Our embedding of SPARQL into a non-monotonic rules language might provide valuable insights here, since it opens up a whole body of work done on combinations of such rules languages with first-order logic based ontology languages, see e.g. (Eiter et al., 2006; Rosati, 2006a; de Bruijn, Eiter, et al., 2007; Motik & Rosati,

2007; de Bruijn, Pearce, Polleres, & Valverde, 2007). As such, with the translation herein, we aim at providing a first step towards a common logical framework where Semantic Web standard play together. As mentioned in the introduction, there are plenty of works on embedding (fragments of) OWL and ASP style rules into a common logical framework; with the present work we hope to have opened the door a bit further towards including SPARQL in this picture. We note that such a common logical framework would open up work on equivalences of SPARQL queries modulo ASP over RDF, plus ASP-expressible fragments of OWL, such as e.g. OWL RL (Motik et al., 2012). Existing work on equivalences of SPARQL queries (Pérez et al., 2009; Schmidt, Meier, & Lausen, 2010; Letelier et al., 2012) or on conjunctive queries for ontologies (Eiter, Lutz, Ortiz, & Simkus, 2009; Glimm, Lutz, Horrocks, & Sattler, 2008; Glimm, Horrocks, & Sattler, 2008; Krötzsch, Rudolph, & Hitzler, 2007b; Glimm & Rudolph, 2010) do not yet cover these combined aspects; an exception here is the work of Melisachew Chekol et al. who suggest to encode parts of OWL and SPARQL into μ-calculus to test query containment in this framework, see (Chekol, Jérôme, & Pierre Genevés, 2012; Chekol, 2013). We believe that looking into ASP – and (quantified) Equilibrium Logic (Pearce & Valverde, 2008) as its underlying logic – could be an interesting alternative path along these lines.

5.4 Implementing SPARQL on top of ASP Engines and Relational Database Systems

We have incorporated an earlier version of our translation within the dlv-DB (Ianni, Krennwallner, Martello, & Polleres, 2009) system, towards a rule-enabled RDF store with SPARQL support. As for related work here, we should not forget to remark that also work by Cyganiak (Cyganiak, 2005), Lu et al. (Lu, Cao, Ma, Yu, & Pan, 2007) who embed SPARQL in more traditional relational algebra, in order to implement SPARQL on top of existing SQL engines. SPARQL access to databases, by the way, is not restricted to the usage of a relational database system as an RDF Store, but so-called RDB-to-RDF mappings based on views or rules can allow to integrate arbitrary relational schemata and legacy databases with the Semantic Web and SPARQL worlds, cf. (Das, Sundara, & Cyganiak, 2012; Arenas, Bertails, Prud'hommeaux, & Sequeda, 2012; Sequeda, Arenas, & Miranker, 2012). Particularly, it should be noted here that (Sequeda et al., 2012) base their mappings on Datalog, which provides a possible connection to our work.

6 Conclusions

In this paper, we extended the formalisation of the SPARQL semantics based on (Pérez et al., 2009; Arenas et al., 2009) and (Polleres et al., 2007) by the new features of SPARQL1.1 and provided a translation to ASP.

In particular, we studied the new features of subqueries, solution modifiers, property paths, negation patterns and assignment. Some of these have been investigated before, in (Angles & Gutierrez, 2011) several types of subqueries were presented and in (Arenas, Conca, & Pérez, 2012)

an in-depth treatment of an earlier version of property paths is shown. We based our formal semantics on these works and on the recent SPARQL1.1 specification (Harris & Seaborne, 2013).

Moreover we collected and extended the discussion of peculiarities in the specification, e.g. the treatment of FILTER expressions and bound or unbound variables, which depend whether the filter occurs as a top level filter in an OPT pattern or if the variable occurs in an EXISTS atomic filter. The newly introduced MINUS pattern is close to usual set difference based on incompatible substitutions, with the subtle difference, that disjoint substitutions are handled as a special case.

We provided a full translation of the SPARQL1.0 features to ASP^{nr} (i.e., the ASP fragment corresponding to non-recursive Datalog with default negation), confirming the results of (Angles & Gutierrez, 2008) for the official W3C semantics. For the translation of solution modifiers, as well as new features in SPARQL1.1 we require more general fragments of ASP. The translation of property paths and solution modifiers needs recursion for the variable length paths and the ordering of solutions. The assignment pattern BIND and CONSTRUCT queries add the additional requirement of external predicates to handle the creation of new values. Strong safety and hence termination is still guaranteed in these programs, as long as we restrict to the translation of single queries, whereas extended RDF graphs (from the discussion in Section 5.2) would possibly involve recursion also potentially over default negation. ASP, as a generalized form of Datalog, augmented with external predicates (such as provided in HEX-programs (Eiter et al., 2005)) can cope with these requirements and therefore is a particular candidate for further investigating foundations and extensions of SPARQL1.1. We left out one feature of SPARQL1.1, namely aggregates, which fall into a similar category as CONSTRUCT queries and those that contain BIND, since such aggregate functions create new values as well. As discussed in (Polleres et al., 2007), one could again use HEX-programs for encoding these, as HEX-programs allow external predicates to inspect a whole interpretation to compute e.g. an average value.

Acknowledgments

This article is based on first results published in the paper titled "From SPARQL to Rules (and back)" (Polleres, 2007) which appeared in the proceedings of the World Wide Web conference 2007. The authors are grateful for reviewer comments for the conference version. Special thanks go to Jos de Bruijn, Giovambattista Ianni, and Reto Krummenacher for discussions on earlier versions of this document, to Bijan Parsia, Jorge Pérez, and Andy Seaborne for valuable insights gained through various email-discussions, and finally to Roman Schindlauer and Thomas Krennwallner for their invaluable help on prototype implementations on top of dlvhex. Moreover, thanks go to all members of the SPARQL1.1 working group. This work was started supported by the Spanish MEC under the project TIC-2003-9001, in the course of which the corresponding author also had the opportunity to collaborate directly with David Pearce, to whom this special issue is dedicated to; therefore, part of our gratitude we could start the work on SPARQL originally also goes to David. This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT12-015.

Bibliography

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Alkhateeb, F., Baget, J.-F., & Euzenat, J. (2009). Extending SPARQL with regular expression patterns (for querying RDF). *Journal of Web Semantics*, *7*(2), 57-73.

Angles, R., & Gutierrez, C. (2008). The expressive power of SPARQL. In A. P. Sheth et al. (Eds.), *International Semantic Web Conference (ISWC 2008)* (Vol. 5318, pp. 114–129). Karlsruhe, Germany: Springer.

Angles, R., & Gutierrez, C. (2011). Subqueries in SPARQL. In P. Barceló & V. Tannen (Eds.), *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011* (Vol. 749). CEUR-WS.org.

Arenas, M., Bertails, A., Prud'hommeaux, E., & Sequeda, J. (2012, September 27). *A direct mapping of relational data to rdf.* (W3C Recommendation, available at `http://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/`)

Arenas, M., Conca, S., & Pérez, J. (2012). Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In A. Mille, F. L. Gandon, J. Misselis, M. Rabinovich, & S. Staab (Eds.), *Proceedings of the 21st world wide web conference 2012 (www2012)* (p. 629-638). ACM.

Arenas, M., Gutierrez, C., & Pérez, J. (2009). On the Semantics of SPARQL. In R. D. Virgilio, F. Giunchiglia, & L. Tanca (Eds.), *Semantic Web Information Management - A Model-Based Perspective* (p. 281-307). Springer.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Beckett, D., & Berners-Lee, T. (2008, January 14). *Turtle - Terse RDF Triple Language.* (W3C Team Submission, `http://www.w3.org/TeamSubmission/turtle/`)

Berners-Lee, T. (1999). *Weaving the web*. Harper.

Berners-Lee, T. (2006). *Linked Data.* (`http://www.w3.org/DesignIssues/LinkedData.html`)

Boley, H., Kifer, M., Pătrânjan, P.-L., & Polleres, A. (2007, September 3–7). Rule interchange on the web. In *Reasoning web 2007* (Vol. 4636, pp. 269–309). Springer.

Brewka, G., Eiter, T., & Truszczynski, M. (2011). Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92-103.

Brickley, D., & Guha, R. (2004, February 10). *RDF vocabulary description language 1.0: RDF Schema.* (W3C Recommendation, available at `http://www.w3.org/TR/rdf-schema/`)

Brickley, D., & Miller, L. (2007, November 2). *FOAF Vocabulary Specification 0.91.* (`http://xmlns.com/foaf/spec/`)

Calimeri, F., & Ianni, G. (2005, September 5–8). External sources of computation for Answer Set Solvers. In C. Baral, G. Greco, N. Leone, & G. Terracina (Eds.), *Proceedings of the 8th*

international conference on logic programming and nonmonotonic reasoning (Vol. 3662, pp. 105–118). Diamante, Italy: Springer Verlag.

Carroll, J., Bizer, C., Hayes, P., & Stickler, P. (2005). Named graphs. *Journal of Web Semantics*, *3*(4), 247–267.

Chekol, M. W. (2013). *Analyse statique de requête pour le web sémantique (static analysis of semantic web queries)*. Unpublished doctoral dissertation. (defended)

Chekol, M. W., Jérôme, & Pierre Genevés, N. L. (2012). Sparql query containment under rdfs entailment regime. In *International joint conference on automated reasoning (ijcar2012)*.

Cyganiak, R. (2005, September 28). *A relational algebra for sparql* (Tech. Rep. No. HPL-2005-170). Bristol, UK: HP Labs.

Dantsin, E., Eiter, T., Gottlob, G., & Voronkov, A. (2001). Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, *33*(3), 374–425.

Das, S., Sundara, S., & Cyganiak, R. (2012, September 27). *R2rml: Rdb to rdf mapping language*. (W3C Recommendation, available at `http://www.w3.org/TR/2012/REC-r2rml-20120927/`)

de Bruijn, J. (2010, June 22). *RIF RDF and OWL Compatibility*. (W3C Recommendation, available at `http://www.w3.org/TR/rif-rdf-owl/`)

de Bruijn, J., Eiter, T., Polleres, A., & Tompits, H. (2007, January 6–12). Embedding Non-Ground Logic Programs into Autoepistemic Logic for Knowledge-Base Combination. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)* (pp. 304–309). Hyderabad, India: AAAI.

de Bruijn, J., Franconi, E., & Tessaris, S. (2005, November). Logical reconstruction of normative RDF. In *OWL: Experiences and directions workshop (OWLED-2005)*. Galway, Ireland.

de Bruijn, J., Pearce, D., Polleres, A., & Valverde, A. (2007, June 7–8). Quantified equilibrium logic and hybrid rules. In M. Marchiori, J. Z. Pan, & C. de Sainte Marie (Eds.), *First international conference on web reasoning and rule systems (rr2007)* (Vol. 4524, pp. 58–72). Innsbruck, Austria: Springer.

de Bruijn, J., Pearce, D., Polleres, A., & Valverde, A. (2010). A semantical framework for hybrid knowledge bases. *Knowledge and Information Systems (KAIS)*, *25*(1), 81–104.

Duerst, M., & Suignard, M. (2005, January). *Internationalized Resource Identifiers (IRIs)* (No. 3987). RFC 3987 (Proposed Standard). IETF. (available at `rfc3987.txt`)

Eiter, T., Ianni, G., & Krennwallner, T. (2009). Answer set programming: A primer. In S. Tessaris et al. (Eds.), *Reasoning web. semantic technologies for information systems, 5th international summer school 2009, brixen-bressanone, italy, august 30 - september 4, 2009, tutorial lectures* (Vol. 5689, p. 40-110). Springer.

Eiter, T., Ianni, G., Krennwallner, T., & Polleres, A. (2008, September 7–11). Rules and ontologies for the semantic web. In C. Baroglio, P. A. Bonatti, J. Maluszynski, M. Marchiori, A. Polleres, & S. Schaffert (Eds.), *Reasoning web 2008* (Vol. 5224, pp. 1–53). San Servolo Island, Venice, Italy: Springer.

Eiter, T., Ianni, G., Polleres, A., Schindlauer, R., & Tompits, H. (2006, September 4–8). Reasoning with rules and ontologies. In *Reasoning web 2006* (Vol. 4126, pp. 93–127). Lisbon, Portugal: Springer.

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

Eiter, T., Lukasiewicz, T., Schindlauer, R., & Tompits, H. (2004). Combining answer set programming with description logics for the semantic web. In *Proceedings of the ninth international conference on principles of knowledge representation and reasoning (kr'04)*. Whistler, Canada: AAAI Press.

Eiter, T., Lutz, C., Ortiz, M., & Simkus, M. (2009, July 11–17). Query answering in description logics with transitive roles. In *Proceedings of the 21st international joint conference on artificial intelligence (ijcai 2009)* (p. 759-764). Pasadena, California, USA.

Faber, W., Leone, N., & Pfeifer, G. (2004, September). Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes & J. Leite (Eds.), *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)* (pp. 200–212). Lisbon, Portugal: Springer Verlag.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). *Answer set solving in practice*. Morgan & Claypool.

Gelder, A. V., Ross, K., & Schlipf, J. S. (1988). Unfounded sets and well-founded semantics for general logic programs. In *7th acm symposium on principles of database systems* (pp. 221–230). Austin, Texas: ACM.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In R. A. Kowalski & K. Bowen (Eds.), *5th int'l conf. on logic programming* (pp. 1070–1080). Cambridge, Massachusetts: The MIT Press.

Gelfond, M., & Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, *9*, 365–385.

Glimm, B., Horrocks, I., & Sattler, U. (2008, September 16–19). Unions of conjunctive queries in shoq. In *Principles of knowledge representation and reasoning: Proceedings of the eleventh international conference, kr 2008* (pp. 252–262). Sydney, Australia: AAAI Press.

Glimm, B., Lutz, C., Horrocks, I., & Sattler, U. (2008). Conjunctive query answering for the description logic shiq. *J. Artif. Intell. Res. (JAIR)*, *31*, 157-204.

Glimm, B., Ogbuji, C., Hawke, S., Herman, I., Parsia, B., Polleres, A., & Seaborne, A. (2013, March 21). *SPARQL 1.1 Entailment Regimes.* (W3C Recommendation, available at `http://www.w3.org/TR/sparql11-entailment/`)

Glimm, B., & Rudolph, S. (2010, May 9–13). Status QIO: Conjunctive query entailment is decidable. In *Proceedings of the 12th international conference on the principles of knowledge representation and reasoning (kr-10)* (pp. 225–235). Toronto, Canada: AAAI Press.

Grosof, B. N., Horrocks, I., Volz, R., & Decker, S. (2003). Description logic programs: Combining logic programs with description logic. In *12th international conference on world wide web (www'03)* (pp. 48–57). Budapest, Hungary: ACM.

Harris, S., & Seaborne, A. (2013, March 21). *SPARQL 1.1 Query Language.* (W3C Recommendation at `http://www.w3.org/TR/sparql11-query/`)

Hayes, P. (2004, February). *RDF semantics.* (W3C Recommendation, available at`http://`

www.w3.org/TR/rdf-mt/)

Heath, T., & Bizer, C. (2011). *Linked data: Evolving the web into a global data space*. Morgan & Claypool.

Ianni, G., Krennwallner, T., Martello, A., & Polleres, A. (2009, October 25–29). Dynamic querying of mass-storage rdf data with rule-based entailment regimes. Washington DC, USA: Springer.

Kifer, M., & Boley, H. (2012, December 11). *RIF Overview.* (W3C Working Group Note, available at http://www.w3.org/TR/rif-overview/)

Kollia, I., Glimm, B., & Horrocks, I. (2011). SPARQL query answering over OWL ontologies. In *8th extended semantic web conference (eswc2011)* (Vol. 6643, pp. 382–396). Springer.

Krisnadhi, A., Maier, F., & Hitzler, P. (2011). OWL and rules. In *Reasoning web 2011* (p. 382-415).

Krötzsch, M., Rudolph, S., & Hitzler, P. (2007a, July 22–26). Complexity boundaries for horn description logics. In *Proceedings of the twenty-second aaai conference on artificial intelligence (aaai)* (pp. 452–457). Vancouver, British Columbia, Canada.

Krötzsch, M., Rudolph, S., & Hitzler, P. (2007b, November 11–15). Conjunctive queries for a tractable fragment of OWL 1.1. In *Proceedings of the 6th international semantic web conference and 2nd asian semantic web conference, iswc 2007 + aswc 2007* (pp. 310–323). Busan, Korea.

Letelier, A., Pérez, J., Pichler, R., & Skritek, S. (2012). Static analysis and optimization of semantic web queries. In *Proceedings of the 31st acm sigmod-sigact-sigart symposium on principles of database systems (pods 2012)* (p. 89-100).

Levy, A. Y., & Rousset, M.-C. (1998). Combining horn rules and description logics in CARIN. *Artificial Intelligence*, *104*, 165–209.

Lifschitz, V. (1999). Answer set planning. In *Iclp* (p. 23-37).

Losemann, K., & Martens, W. (2012, May 20–24). The complexity of evaluating path expressions in sparql. In *Proceedings of the 31st acm sigmod-sigact-sigart symposium on principles of database systems, (pods 2012)* (p. 101-112). Scottsdale, AZ, USA: ACM.

Lu, J., Cao, F., Ma, L., Yu, Y., & Pan, Y. (2007). An Effective SPARQL Support over Relational Databases. In *Swdb-odbis* (p. 57-76).

Lukasiewicz, T. (2010). A novel combination of answer set programming with description logics for the semantic web. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. (In press)

Manola, F., & Miller, E. (2004, February 10). *RDF primer.* (W3C Recommendation, available at http://www.w3.org/TR/rdf-primer/)

Marek, V. W. (1999). Stable models and an alternative logic programming paradigm. In *In the logic programming paradigm: a 25-year perspective* (pp. 375–398). Springer-Verlag.

Motik, B., Grau, B. C., Horrocks, I., Wu, Z., Fokoue, A., & Lutz (eds.), C. (2012, December 11). *OWL 2 web ontology language profiles (second edition).* (W3C Recommendation, available at http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/)

Motik, B., & Rosati, R. (2007). A faithful integration of description logics with logic programming. In *Twentieth international joint conference on artificial intelligence (ijcai'07)* (pp.

477–482). Hyderabad, India: AAAI.

Motik, B., Sattler, U., & Studer, R. (2005). Query answering for OWL-DL with rules. *Journal of Web Semantics*, *3*(1), 41–60.

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, *25*(3-4), 241-273.

Niemelä, I., Simons, P., & Soininen, T. (1999). Stable model semantics of weight constraint rules. In M. Gelfond, N. Leone, & G. Pfeifer (Eds.), *Lpnmr* (Vol. 1730, p. 317-331). Springer.

Pearce, D., & Valverde, A. (2008). Quantified equilibrium logic and foundations for answer set programs. In *24th international conference on logic programming (iclp2008)* (p. 546-560).

Pérez, J., Arenas, M., & Gutierrez, C. (2006, May 26). Semantics and complexity of SPARQL. In *Iswc 2006, 5th international semantic web conference* (Vol. 4273, pp. 30–43). Springer.

Pérez, J., Arenas, M., & Gutierrez, C. (2008). nSPARQL: A navigational language for RDF. In *7th international semantic web conference, iswc 2008* (Vol. 5318, pp. 66–81). Springer.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, *34*(3), Article 16 (45 pages).

Polleres, A. (2007, May 8–12). From SPARQL to rules (and back). In *Proceedings of the 16th world wide web conference (www2007)* (pp. 787–796). Banff, Canada: ACM Press.

Polleres, A. (2013, January). Agreement technologies and the semantic web. In S. Ossowski (Ed.), *Agreement technologies* (Vol. 8, pp. 57–68). Springer. (To appear)

Polleres, A., Scharffe, F., & Schindlauer, R. (2007, November 27–29). SPARQL++ for mapping between RDF vocabularies. In *Otm 2007, part i : Proceedings of the 6th international conference on ontologies, databases, and applications of semantics (odbase 2007)* (Vol. 4803, pp. 878–896). Vilamoura, Algarve, Portugal: Springer.

Polleres, A., & Schindlauer, R. (2007, September 13). dlvhex-sparql: A SPARQL-compliant query engine based on dlvhex. In *2nd international workshop on applications of logic programming to the web, semantic web and semantic web services (alpsws2007)* (Vol. 287, pp. 3–12). Porto, Portugal: CEUR-WS.org.

Prud′hommeaux, E., & Seaborne, A. (2008, January 15). *SPARQL Query Language for RDF*. (W3C Recommendation, available at `http://www.w3.org/TR/rdf-sparql-query/`)

Przymusinski, T. C. (1988). On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming* (pp. 193–216). Morgan Kaufmann Publishers, Inc.

Rosati, R. (2005a). On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, *3*(1), 61–73.

Rosati, R. (2005b). Semantic and computational advantages of the safe integration of ontologies and rules. In *Proceedings of the third international workshop on principles and practice of semantic web reasoning (ppswr 2005)* (Vol. 3703, pp. 50–64). Springer.

Rosati, R. (2006a, September 4–8). Integrating Ontologies and Rules: Semantic and Computational Issues. In P. Barahona, F. Bry, E. Franconi, U. Sattler, & N. Henze (Eds.), *Reasoning web, second international summer school 2006, lissabon, portugal, september 25-29, 2006, tutorial lectures* (Vol. 4126, pp. 128–151). Springer.

Rosati, R. (2006b). $\mathcal{DL} + log$: Tight integration of description logics and disjunctive datalog. In *Proceedings of the tenth international conference on principles of knowledge representation and reasoning (kr'06)* (pp. 68–78).

Schenk, S., & Staab, S. (2007). *Networked rdf graphs* (Tech. Rep.). Koblenz, Germany: Universität Koblenz-Landau. (`http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf`)

Schenk, S., & Staab, S. (2008). Networked graphs: A declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In *Proceedings www-2008* (pp. 585–594). Beijing, China: ACM Press.

Schindlauer, R. (2006). *Answer-set programming for the semantic web*. Unpublished doctoral dissertation, Vienna University of Technology.

Schmidt, M., Meier, M., & Lausen, G. (2010, March 22–25). Foundations of SPARQL query optimization. In *13th international conference on database theory (icdt2010).* Lausanne, Switzerland.

Sequeda, J., Arenas, M., & Miranker, D. P. (2012). On directly mapping relational databases to RDF and OWL. In *Proceedings of the 21st world wide web conference 2012 (www2012)* (p. 649-658).

Smith, M. K., Welty, C., & McGuinness, D. L. (2004, February 10). *OWL Web Ontology Language Guide.* (W3C Recommendation, available at `http://www.w3.org/TR/owl-guide/`)

SQL-99. (1999, October 1). *Information Technology - Database Language SQL- Part 3: Call Level Interface (SQL/CLI)* (Tech. Rep. No. INCITS/ISO/IEC 9075-3). INCITS/ISO/IEC. (Standard specification)

Ullman, J. D. (1989). *Principles of Database and Knowledge Base Systems*. New York, NY, USA: Computer Science Press.

W3C OWL 2 Working Group. (2012, December 11). *OWL 2 Web Ontology Language Document Overview (Second Edition).* (W3C Recommendation, available at `http://www.w3.org/TR/owl2-overview/`)

Appendix: Translations of Sample Queries

In this appendix we provide corresponding programs and answers for some sample queries mentioned in this paper in order to exemplify the translation. We will, for each query, first give the query in SPARQL syntax as specified in (Harris & Seaborne, 2013), then provide the pattern in our notation and, finally provide the translated logic programs.

For reasons of clarity, we explicitly specify the dataset in the form of FROM and FROM NAMED clauses in SPARQL syntax. As before, we omit and the leading 'http://' or other schema identifiers in IRIs. Note that we omit the ASP rules for solution modifiers in these example translations for readability, except for the translation of Example 2.6. Therefore we will include the translation of the SELECT clause also only in the last translation of Example 2.6, since the projection of the SELECT clause is included in the solution modifiers.

Query 1

We start with the query from Figure 1.

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

This corresponds to $Q_1 = (DS_1, P_1, SM_1)$ with $DS_1 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, $SM_1 = ((), \{?X, ?Y\}, false, 0, 0)$, and

$$P_1 = (?Y,\ \texttt{foaf:name},\ ?X)$$

$\Pi_{Q_1} = \Pi^{DS_1}_{\mathsf{FILTER}} \cup$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
ans1(X,Y,default) :- triple(Y,foaf:name,X,default).
```

The query delivers the following answers:

```
{ ans1("Bob",_:a,default),
  ans1("Bob",_:c,default),
  ans1("Alice","alice.org#me",default) }
```

Query 2

We continue with the second query from Example 2.3.

```
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?N
FROM <ex.org/bob>
WHERE { ?A foaf:knows+/foaf:name ?N .
        ?A foaf:name "Alice"}
```

This corresponds to $Q_2 = (DS_2, P_2, SM_2)$ with $DS_2 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, $SM_2 = ((), \{?N\}, false, 0, 0)$, and

$$P_2 = ((?A,\ \texttt{foaf:knows} + /\texttt{foaf:name},\ ?N)\ \mathsf{AND}\ (?A,\ \texttt{foaf:name},\ "Alice"))$$

$\Pi_{Q_1} = \Pi^{DS_1}_{\mathsf{FILTER}} \cup Join(1) \cup$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
ans1(A,N,S1,default) :- ans2(A',N',S1',default),
                        ans3(A'',default),
                        join1(A',A'',A).
ans2(A,N,S1,default) :- ans4(A,S1',default),
                        ans5(N,S1'',default),
                        join1(S1',S1'',S1).
ans3(A,default) :- triple(A,foaf:name,"Alice",default).
ans4(A,S1,default) :- reach4(A,S1,default).
reach4(X4,Y4,default) :- ans8(X4,Y4,default).
reach4(Z4,Y4,default) :- reach4(Z4,X4,default),
                         ans8(X4,Y4,default).
ans5(N,S1,default) :- triple(S1,foaf:name,N,default).
ans8(X4,Y4,default) :- triple(X4,foaf:knows,Y4,default).
```

The query delivers the following answer:

```
{ ans1(_:b,"Bob",_:c,default)}
```

Query 3

We continue with the query from Example 2.4.

```
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?N
FROM <alice.org>
WHERE { ?P foaf:name ?N .
        FILTER ?G ( EXISTS (?P foaf:knows ?F .
                            ?F foaf:name "Bob")) }
```

This corresponds to $Q_3 = (DS_3, P_3, SM_3)$ with $DS_3 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, $SM_3 = ((), \{?N\}, false, 0, 0)$, and

$P_3 = ((?P,\ \texttt{foaf:name}, ?M)$
FILTER (EXISTS$((?P, \texttt{foaf:knows}, ?F)$ AND $(?F, \texttt{foaf:name}, "Bob"))))$

$\Pi_{Q_3} = \Pi^{DS_3}_{\mathsf{FILTER}} \cup Join(1) \cup$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
ans1(M,P,default) :- ans2(P,M,default), filter1_1(P,M,true).
ans2(M,P,default) :- triple(P,foaf:name,N,default).
filter1_1(M',P',true) :- ans3_1(F,P,M',P',default).
filter1_1(M,P,false) :- ans2(M,P,default),
                        not filter1_1(M,P,true).
ans3_1(F,P,M21,P21,default) :- ans6_1(F',P,M21,P21,default),
                               ans7_1(F'',M21,P21,default),
                               join1(F',F'',F).
ans6_1(F,P,M,P',default) :- triple(F,foaf:knows,P,default),
                            ans2(M21,P21,default),
                            join1(P,P21,P21s).
ans7_1(F,M21,P21,default) :-
                          triple(F,foaf:name,"Bob",default),
                          ans2(M21,P21,default).
```

The query delivers the following answer:

```
{ ans1("Alice",_:b,default) }
```

Query 4

We continue with the query from Example 2.5.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?X
FROM <alice.org>
WHERE { ?P foaf:nick ?N .
        ?P foaf:name ?M .
        BIND concat(?M," a.k.a. ",?N) AS ?X}
```

This corresponds to $Q_4 = (DS_4, P_4, SM_4)$ with $DS_4 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, $SM_4 = ((), \{?X\}, false, 0, 0)$, and

$P_4 = (((?P,\ \texttt{foaf:nick},\ ?N)\ \mathsf{AND}\ (?P, \texttt{foaf:name}, ?M))$
$\quad \mathsf{BIND}\ concat(?M, "a.k.a.", ?N)\ \mathsf{AS}\ ?X$

$\Pi_{Q_4} = \Pi^{DS_4}_{\mathsf{FILTER}} \cup Join(1) \cup$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
ans1(M,N,P,X,default) :-
                  eval["concat(?1,\" a.k.a. \",?2)",M,N](X),
                  ans2(M,N,P,default).
ans2(M,N,P,default) :- ans3(N,P',default),
                       ans4(M,P'',default),
                       join1(P',P'',P).
ans3(N,P,default) :- triple(P,foaf:nick,N,default).
ans3(M,P,default) :- triple(P,foaf:name,M,default).
```

The query delivers the following answer:

```
{ ans1("Bob","Bobby",_:c,"Bob a.k.a. Bobby",default) }
```

Query 5

We continue with the second query from Example 2.6.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?N
FROM <alice.org>
FROM <ex.org/bob>
WHERE { { ?P foaf:name ?M . } UNION { ?P foaf:nick ?N . } }
ORDER BY ?N OFFSET 2 LIMIT 1
```

This corresponds to $Q_5 = (DS_5, P_5, SM_5)$ with $DS_5 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, $SM_5 = ((?N), \{?N\}, true, 2, 1)$, and

$$P_5 = (((?P,\ \texttt{foaf:name},\ ?M)\ \mathsf{UNION}\ (?P, \texttt{foaf:nick}, ?N))$$

$\Pi_{Q_5} = \Pi^{DS_5}_{\mathsf{FILTER}} \cup Join(1) \cup$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
ans2(M,null,P,default) :- ans4(M,P,default).
ans2(null,N,P,default) :- ans5(N,P,default).
ans4(M,P,default) :- triple(P,foaf:name,M,default).
ans5(N,P,default) :- triple(P,foaf:nick,N,default).
```

```
lt2(M,N,P,M',N',P',U1',default) :-
              ans2(M,N,P,default),
              ans2(M',N',P',default),
              N < N'.
lt2(M,N,P,M',N,P',default) :-
              ans2(M,N,P,default),
              ans2(M',N,P',default),
              M < M'.
lt2(M,N,P,,M,N,P',,default) :-
              ans2(M,N,P,default),
              ans2(M,N,P',default),
              P < P'.
remove2(M,N,P,default) :- lt2(M',N,P',M,N,P,default).
notsucc2(M,N,P,M',N',P',default) :-
              lt2(M,N,P,M',N',P',default)
              lt2(M',N',P',M'',N'',P'',default),
              not remove2(M',N',P',default).
succ2(M,N,P,M',N',P',default) :-
              lt2(M,N,P,M',N',P',default),
              not notsucc2(M,N,P,M',N',P',default),
              not remove2(M,N,P,default).
              not remove2(M',N',P'',default).
notinf2(M,N,P,default) :-
              lt2(M',N',P',M,N,P,default),
              not remove2(M',N',P',default).
inf2(M,N,P,default) :-
              ans2(M,N,P,default),
              not notinf2(M,N,P,default),
              not remove2(M,N,P,default).
anso2(M,N,P,default,1) :- inf2(M,N,P,default).
anso2(M,N,P,default,O) :-
              anso2(M',N',P',default,O'),
              succ2(M',N',P',M,N,Pdefault),
              O = O' + 1.
ans1(N,default) :- anso2(M,N,P,default,O), 2<O, O<=2+1.
```

The query delivers the following answer:

```
{ ans1("Bob",default) }
```

Note that here we also see an example with the projection of the SELECT clause.