Chapter 1
SPARQLog: SPARQL with Rules and Quantification

François Bry, Tim Furche, Bruno Marnette, Clemens Ley, Benedikt Linse, and Olga Poppe

Abstract SPARQL has become the gold-standard for RDF query languages. Nevertheless, we believe there is further room for improving RDF query languages. In this chapter, we investigate the addition of rules and quantifier alternation to SPARQL. That extension, called SPARQLog, extends previous RDF query languages by arbitrary quantifier alternation: blank nodes may occur in the scope of all, some, or none of the universal variables of a rule. In addition SPARQLog is aware of important RDF features such as the distinction between blank nodes, literals and IRIs or the RDFS vocabulary. The semantics of SPARQLog is closed (every answer is an RDF graph), but lifts RDF's restrictions on literal and blank node occurrences for intermediary data. We show how to define a sound and complete operational semantics that can be implemented using existing logic programming techniques. While SPARQLog is Turing complete, we identify a decidable (in fact, polynomial time) fragment SwARQLog ensuring polynomial data-complexity inspired from the notion of super-weak acyclicity in data exchange. Furthermore, we prove that SPARQLog with no universal quantifiers in the scope of existential ones ($\forall\exists$ fragment) is equivalent to full SPARQLog in presence of graph projection. Thus, the convenience of arbitrary quantifier alternation comes, in fact, for free. These results, though here presented in the context of RDF querying, apply similarly also in the more general setting of data exchange.

Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
`http://www.pms.ifi.lmu.de/` · Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
http://web.comlab.ox.ac.uk/oucl/people/clemens.ley.html

1.1 Introduction

Access to data in a machine-processable, domain-independent manner plays a central role in the future growth of the Internet. Information on legislative proceedings, census data, scientific experiments and databases, as well as the data gathered by social network applications is now accessible in form of RDF data. The Resource Description Framework (RDF) is a data format for the Web with a formal semantics that is achieving considerable popularity. Compared to relational databases, RDF is mostly distinguished by (1) a specialization to ternary statements or "triples" relating a subject, via a predicate, to an object, (2) the presence of blank nodes that allow statements where subject or object are unknown, and (3) specific semantics for a small, predefined vocabulary (RDF Schema, or RDFS) reminiscent of an object-oriented type system.

With the staggering amount of data available in RDF form on the Web, the second indispensable ingredient becomes the easy selection and processing of RDF data. For that purpose, a large number of RDF query languages (see [9] for a recent survey) has been proposed, with SPARQL [19] the most prominent representative. In this paper, we build on SPARQL to remedy two of the most significant weaknesses of SPARQL from our perspective: SPARQLog extends SPARQL to support the distinguishing features of RDF such as blank nodes and the logical core [15] of the RDFS vocabulary. More technically speaking, we extend SPARQLog with rules and quantifier alternation. In SPARQLog, Blank nodes can be constructed by existentially quantified variables in rule heads. It allows *full alternation* between existential and universal quantifiers in a rule. This sharply contrasts with previous approaches to rule-based query languages that either do not support blank nodes (in rule heads) at all [18, 23], or only a limited form of quantifier alternation [25, 21, 10].

To illustrate the benefits of full quantifier alternation, imagine an information system about university courses. We distinguish three types of rules with existential quantifiers (and thus blank nodes) based on the alternation of universal and existential quantifiers:
(1) "Someone knows each professor" can be represented in SPARQLog as

```
PREFIX uni: <http://example.org/uni>
FROM   <http://lmu.de/staff/>

EX ?pers ALL ?prof
CONSTRUCT { ?pers foaf:knows ?prof }
WHERE     { ?prof rdf:type uni:professor }
```

We call such rules ∃∀ rules (i.e., rules in the Bernays-Schönfinkel class). Some approaches such as [25] are limited to rules of this form. We show that a recursive rule language that is limited to these kind of rules is strictly less expressive than a language that allows rules also of the form discussed under (2) and (3). The gain is that languages with only ∃∀ rules are still decidable. However, as shown in Section 1.5.1.1, there are larger fragments of SPARQLog that are still decidable.
(2) Imagine, that we would like to state that each lecture must be "practiced" by another course (such as a tutorial or practice lab) without knowing more about that

course. This statement can not be expressed by $\exists\forall$ rules. In SPARQLog it can be represented as

```
PREFIX uni: <http://example.org/uni>
FROM   <http://lmu.de/staff/>

ALL ?lec EX ?crs
CONSTRUCT { ?crs uni:practices ?lec }
WHERE     { ?lec rdf:type uni:lecture }
```

Such rules are referred to as $\forall\exists$ rules (sometimes also denoted as $\forall^*\exists^*$ rules). Recent proposals for rule extensions to SPARQL are limited to this form, if they consider blank nodes in rule heads at all. The reason is that in SPARQL `CONSTRUCT` patterns a fresh blank node is constructed for every binding of the universal variables (see Section 10.2.1 in [19]). For a more detailed comparison of SPARQL and SPARQLog, see Sections 1.4 and 1.3.

(3) To the best of our knowledge, SPARQLog is the first RDF query language that supports the third kind of rules, where quantifiers are allowed to alternate freely: This allows to express statements such as, for each lecture there is a course that "practices" that lecture and is attended by all students attending the lecture. This is represented in SPARQLog as

```
PREFIX uni: <http://example.org/uni>
FROM   <http://lmu.de/staff/>

ALL ?lec EX ?crs ALL ?stu
CONSTRUCT { ?crs uni:practices ?lec . ?stu uni:attends ?crs }
WHERE     { ?lec rdf:type uni:lecture . ?stu uni:attends ?lec }
```

In Section 1.5.2, we show (for the first time) that rules with full quantifier alternation can be normalized to $\forall\exists$ form if we allow triple projection (more precisely, if we consider only the default graph in the RDF dataset as semantic of a SPARQLog program). Thus full quantifier alternation does not add to the expressiveness of SPARQLog under default-graph semantics. Rather, for all languages with $\forall\exists$ rules and triple projection the rewriting in Section 1.5.2 allows arbitrary quantifier alternation to be added for free.

In addition to flexible support for existential information through full quantifier alternation, SPARQLog captures the essentials of RDF through two further characteristics: First, SPARQLog is a closed RDF query language, i.e., the answer to an SPARQLog program is again an RDF dataset. Second, SPARQLog can express the logical core of the RDFS semantics (ρdf from [15]).

In particular, we follow RDF in allowing blank nodes not in predicate position for answers (as well as literals only in object position). We show that these limitations make the traditional approach of defining a closed semantics for a rule based query language as initial models unpractical. Nonetheless, we show how a closed semantics of a rule based query language for RDF can be defined that captures the consequences of the program under RDF entailment. A consequence of that semantics is that intermediary data, but only intermediary data, may violate the limitations posed by RDF (see also [24]).

With this semantics SPARQLog is unsurprisingly Turing complete. Therefore, we also consider fragments of SPARQLog that are decidable in polynomial time. We

(slightly) extend results from [13] to also cover quantifier alternation and identify a tractable fragment, called SwARQLog. It is based on the notion of super-weak acyclicity from [13] (which is itself inspired from, though strictly more general than, the notion of *weak-acyclicity* in data-exchange [6]). SwARQLog also remains strictly more expressive than restrictions of SPARQLog to $\exists\forall$ rules as in [25].

Contributions. The paper is organised along the following contributions:

1. An extension of SPARQL with *rules and free quantifier alternation*, called SPARQLog is introduced in Section 1.4.
2. The semantics of SPARQLog is defined in terms of entailment in Section 1.4.2. We show how this semantics can be implemented by a *reduction to the evaluation of a standard logic program* without existential quantifiers in Section 1.4.3.
3. SPARQLog is shown to be Turing-complete, but a significant *decidable fragment* is identified in Section 1.5.1.1.
4. A rewriting for SPARQLog programs to reduce quantifier alternation to $\forall\exists$ form, i.e., rules where no universal quantifier occurs in the scope of an existential one, is given in Section 1.5.2 and shown to be equivalent under default-graph semantics. It is worth emphasizing that this rewriting is not possible in general first-order logic, unless we allow an extension of the vocabulary with helper constants that are not part of the semantics of the program. The latter is provided by the default-graph semantics of SPARQLog.
5. The experimental evaluation of a basic prototype shows that the reduction to standard logic programming easily competes with existing SPARQL engines even when considering only the restricted fragment of SPARQLog equivalent to SPARQL, see Section 1.5.3.

The results in this chapter are partially based on previous results on RDFLog, a Datalog extension with quantifier alternation, see [3, 2].

1.2 Preliminaries

In this paper, we adopt the notions of RDF vocabulary, RDF graph, (simple) RDF interpretation, and (simple) RDF entailment from [11].

Definition 1.1 (RDF Graph [11]). An *RDF vocabulary* V consists of two disjoint sets called *IRIs* U and *literals* L. The *blank nodes* B is a set disjoint from U and L. An *RDF graph* is a set of RDF triples where an *RDF triple* is an element of $(\mathsf{U}\cup\mathsf{B})\times\mathsf{U}\times(\mathsf{U}\cup\mathsf{L}\cup\mathsf{B})$. If $t=(\mathsf{s},\mathsf{p},\mathsf{o})$ is an RDF triple then s is the *subject*, p is the *predicate*, and o is the *object* of t.

The set L of literals consists of three subsets, *plain literals*, *typed literals* and *literals with language tags*. In this work we consider only plain literals (and thus drop IL, the interpretation function for typed literals, see Section 1.3 in [11], in the following definitions).

Definition 1.2 (RDF Interpretation [11]). An *interpretation* I of an RDF vocabulary $\mathsf{V} = (\mathsf{U}, \mathsf{L})$ is a tuple $(\mathsf{IR}, \mathsf{LV}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS})$ where IR is a non-empty set of *resources* such that $\mathsf{L} \subseteq \mathsf{LV} \subseteq \mathsf{IR}$, IP is a set of *properties* and $\mathsf{IEXT} : \mathsf{IP} \to 2^{\mathsf{IR}\times\mathsf{IR}}$, and $\mathsf{IS} : \mathsf{U} \to \mathsf{IR} \cup \mathsf{IP}$ are mappings.

RDF assigns a special meaning to a predefined vocabulary, called RDFS (RDF Schema) vocabulary. For example it is required that $\mathsf{IEXT}(\mathsf{IP}(\mathsf{rdfs{:}subPropertyOf}))$ is transitive and reflexive. The formulation of theses constraints on RDF interpretation makes use of a notion of a *class*. We have omitted this notion in the definition above for simplicity. The logical core of RDFS has been identified in [15], denoted as ρdf. An RDF interpretation I is a ρdf *interpretation* if I satisfied the constraints specified in Definition 3 in [15]. ρdf entailment is the same as RDF entailment, but assigns specific semantics to the RDFS vocabulary (e.g., transitivity of $\mathsf{rdf{:}subClassOf}$).

Definition 1.3 (Interpretation of an RDF Graph [11]). Let I be the RDF (ρdf) interpretation $(\mathsf{IR}, \mathsf{LV}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS})$ and $\mathsf{A} : \mathsf{B} \to \mathsf{IR}$ a mapping. Then $[\mathsf{I}+\mathsf{A}](\mathsf{e}) = \mathsf{a}$ if e is the literal a, $[\mathsf{I}+\mathsf{A}](\mathsf{e}) = \mathsf{IS}(\mathsf{e})$ if e is a IRI, $[\mathsf{I}+\mathsf{A}](\mathsf{e}) = \mathsf{A}(\mathsf{e})$ if e is a blank node, and $[\mathsf{I}+\mathsf{A}](\mathsf{e}) = \text{true}$ if $\mathsf{e} = (\mathsf{s},\mathsf{p},\mathsf{o})$ is an RDF triple over V, $\mathsf{I}(\mathsf{p}) \in \mathsf{IP}$ and $(\mathsf{I}(\mathsf{s}), \mathsf{I}(\mathsf{o})) \in \mathsf{IEXT}(\mathsf{I}(\mathsf{p}))$. Finally $\mathsf{I}(\mathsf{g}) = \text{true}$ if there is a mapping $\mathsf{A} : \mathsf{B} \to \mathsf{IR}$ such that $[\mathsf{I}+\mathsf{A}](t) = \text{true}$ for all RDF triples $\mathsf{t} \in \mathsf{g}$.

The semantics of RDF is completed by the notion of entailment: An RDF graph g *RDF-entails* (*ρdf-entails*) an RDF graph h if for all RDF (ρdf) interpretations I, $\mathsf{I}(\mathsf{h}) = \text{true}$ if $\mathsf{I}(\mathsf{g}) = \text{true}$ [11]. This is equivalent to saying that there is a homomorphism from g to h.

We extend the notion of RDF graph to an RDF dataset as in SPARQL. In [19] an RDF dataset is defined as a set of RDF graphs each associated with an identifying IRI of which one is marked as the default graph. Here, we choose a formalization of RDF dataset close to an RDF graph that simplifies latter notation, but captures the same intuition.

Definition 1.4 (RDF Dataset). An RDF dataset D is a set of quadruples from $(\mathsf{U} \cup \mathsf{B}) \times \mathsf{U} \times (\mathsf{U} \cup \mathsf{L} \cup \mathsf{B}) \times (\mathsf{U} \cup \{\diamond\})$ such that for all (s,p,o,g), (s',p',o',g') it holds that if $\{s,o\} \cap \{s',o'\} = b \neq \emptyset$ and $b \in \mathsf{B}$ then $g = g'$.

Thus, an RDF dataset is a set of triples each extended with an IRI or $\diamond$ that indicates the provenance of the triple from one RDF graphs. Triples from two distinct RDF graphs in a same RDF dataset may not share any blank node. In other words, A RDF dataset is a set of extended RDF triples, where extended means that each triple is assigned a provenance in the form of the name of an RDF graph. A RDF dataset requires that if the same blank node occur in two extended triples of the dataset, then these triples have the same provenance. $\diamond$ indicates that the triple occurs in the default graph, otherwise the IRI identifies the named graph the triple originates from.

For an RDF dataset D, we denote with $\mathsf{D}[g] = \{(s,p,o) : \exists (s,p,o,g) \in \mathsf{D}\}$ the triples (without the graph identifier) in D that belong to the RDF graph with IRI $g \in \mathsf{U} \cup \{\diamond\}$.

We can lift the notion of RDF-entailment to RDF datasets as follows: An RDF dataset D *RDF-entails* (*ρdf-entails*) an RDF dataset E iff for all $g \in IRI \cup \{\diamond\}$ $\mathsf{D}[g]$ RDF-entails $\mathsf{E}[g]$.

For the semantics of SPARQLog, we make use of the following mapping from RDF graphs and datasets to first-order formulas:

Definition 1.5 (Canonical Formula). For an RDF graph $\mathsf{g} = \{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$ we define the canonical formula

$$\phi(\mathsf{g}) = \exists b_1 \ldots \exists b_m (R(s_1, p_1, o_1) \wedge \ldots \wedge R(s_n, p_n, o_n))$$

where $b_1, \ldots, b_m$ are the blank nodes occurring in g and R is a fixed ternary relation symbol.

For an RDF dataset D containing the graph identifiers $u_1, \ldots, u_n$ we define the canonical formula

$$\phi(\mathsf{D}) = \bigwedge_{1 \leq i \leq n} \exists b_1^{u_i} \ldots \exists b_{m_{u_i}}^{u_i} (\bigwedge_{(s,p,o,u_i) \in \mathsf{D}} R(s, p, o, u_i))$$

where $b_1^{u_i}, \ldots, b_{m_{u_i}}^{u_i}$ are the blank nodes occurring in the graph with identifier u_i and R is a fixed relation symbol with arity 4.

It is worth noting (and easy to prove) that the notion of RDF entailment coincides with first-order entailment on the canonical formulas of RDF graphs, resp. RDF datasets.

Lemma 1.1. *Let g, h be RDF graphs (datasets). Then g RDF-entails h iff $\phi(\mathsf{g})$ FO-entails $\phi(\mathsf{h})$.*

1.3 SPARQL Rule Languages

1.3.1 SPARQL and Rule Extensions of SPARQL

As briefly outlined above, SPARQLog is distinguished from previous RDF query languages by the support for arbitrary quantifier alternation. SPARQL [19], which is quickly becoming the yardstick for RDF query languages, supports only $\forall\exists$ quantification in what corresponds to rule heads: Each blank node in a `CONSTRUCT` clause is instantiated once for each binding tuple of the (universal) variables in that clause. Otherwise, SPARQLog and SPARQL queries are roughly equivalent with the exception of negation and typed literals that are not supported in SPARQLog. Furthermore, SPARQL only considers what amounts to a single (non-recursive) rule.

There have been several proposals [18, 21] for extending SPARQL with multiple rules. Typically these either explicitly do not deal with blank nodes in `CONSTRUCT` clauses as [18] or consider only $\forall\exists$ quantification as in basic SPARQL [21]. Their

semantics also differs considerably from SPARQLog as they support negation with answer-set or well-founded semantics. It is worth noting that the characterization of the decidable class of super-weakly acyclic programs as well as the $\forall\exists$ rewriting carry over to SPARQLog with negation fairly immediately.

As SPARQLog [21] consider querying not a single RDF graph but a set of named RDF graphs with may contain dynamically computed views. The authors extend named RDF graphs to so called *networked graphs* allowing:

1. "reuse of RDF graphs enabling the dynamic copying of contents from one graph to the other,
2. viewing RDF graphs in a way that is defined by another RDF graph and
3. dynamic networking of RDF graphs. RDF graphs constitute databases and the meaning they describe comes from their dynamic networking" [21].

In a sense, networked graphs are a mixture of SPARQL datasets and SPARQLog programs.

Definition 1.6 (Networked Graph, adopted from [21]). A *networked graph* $G_N = (N, G, [G_1, ..., G_n], v)$ is encoded in a named RDF graph with name N where G is an RDF graph containing the explicit triples to be included in G_N. $[G_1, ..., G_n]$ is a list of networked graphs and v a mapping from that list of networked graphs to an RDF graph called the view definition of G_N. The *view definition* is included in G_N by statements of the form:

```
N g:definedBy "query".
```

where the prefix g is appropriately bound and `"query"` a literal containing a CONSTRUCT rule. We call the literal a *sub-query* of the networked graph definition. The view definition is the union of the sub-queries.

The following example from [21] illustrates the notion of networked graph. We assume that the named RDF graph ISWebGraph contains information about researchers working at the Information Systems and Semantic Web lab of the Institute of computer science (IFI) and the named RDF graph IFIAdminGraph information about the administrative staff at IFI. Then the named graph u:IFIGraph shown below is a networked graph: (u:IFIGraph, { u:ISWeb u:workingGroupOf u:IFI . u:IFI u:belongsTo u:CSDepartment }, [ISWebGraph, IFIAdminGraph], v). v maps the RDF named graph IFIAdminGraph to itself and the RDF named graph ISWebGraph to an RDF graph about persons that work at `u:IFI` if they are known to work at u:ISWeb.

```
u:IFIGraph {
 u:ISWeb u:workingGroupOf u:IFI . u:IFI u:belongsTo u:CSDepartment .
 u:IFIGraph g:definedBy "CONSTRUCT { ?s ?p ?o }
                         FROM NAMED u:IFIAdminGraph
                         WHERE { GRAPH u:IFIAdminGraph { ?s ?p ?o } }"
 u:IFIGraph g:definedBy "CONSTRUCT { ?person u:worksAt u:IFI }
                         FROM NAMED u:ISWebGraph
                         WHERE { GRAPH u:ISWebGraph {
                                   ?person u:worksAt u:ISWeb. }" }
```

The advantage of this approach is the ability to encode the view definitions directly into RDF graphs where their definitions can also be processed by RDF tools that are not networked aware.

In SPARQLog, we can provide the same definition (using `u:IFIGraph` as default graph). However, SPARQLog also allows multiple different named graphs as targets for the same construction. More importantly, SPARQLog provides full quantifier alternation and has a far simpler semantics (admittedly partly due to the absence of negation):

```
FROM NAMED u:IFIAdminGraph
FROM NAMED u:ISWebGraph

CONSTRUCT { u:ISWeb u:workingGroupOf u:IFI.
            u:IFI u:belongsTo u:CSDepartment }

ALL ?s ?p ?o
CONSTRUCT { ?s ?p ?o }
WHERE     GRAPH u:IFIAdminGraph { ?s ?p ?o }

ALL ?person
CONSTRUCT { ?person u:worksAt u:IFI }
WHERE     GRAPH u:ISWebGraph { ?person u:worksAt u:ISWeb. }
```

1.3.2 Other Rule-based RDF Query Languages

The other class of recursive, rule-based query languages for RDF are adaptations of F-Logic for RDF: As in the case of extending SPARQL, these often do not consider blank nodes in rule heads at all [23]. To the best of our knowledge, the only *decidable* rule-based RDF query language with *blank nodes in rule heads* has been proposed in [25]. Decidability is obtained by restricting rules to $\exists\forall$ form. In this paper, we show that a much less restrictive (and strict super-) class of SPARQLog programs is still decidable, viz. super-weakly acyclic SPARQLog.

Most other RDF query languages such as RQL [12], [10], or SeRQL [1] are limited to what amounts to single SPARQLog rules and do not treat issues such as rule chaining, query closure (i.e., that an answer to a query is again an RDF graph or dataset), and arbitrary quantifier alternation.

SPARQLog shares some similarity with rule extensions for description logics. [20] gives an overview over the limits and possibilities of combining description logics and datalog with and without negation, thereby pointing out minimal undecidable combinations of the two methodologies. Due to the possibility of deriving concepts from base concepts, concept and role inclusion axioms, which are not present in SPARQLog, this problem is fundamentally different and harder to tackle. In particular, the undecidability results from [20] do not carry over to SPARQLog. Moreover such approaches either disallow existential quantifiers in *rule heads* (but allow them in facts or TBox-Axioms), or use full logic programming with function symbols and negation as in dlvhex.

1.3.3 Quantifier Alternation in Data Exchange

As discussed in more detail in Chapter 11, existential quantification and its expressiveness have been extensively studied in data exchange [6]. A data exchange setting consists of two database schemata, a source schema S and the target schema T, and a set of constraints Σ. The data exchange problem is to find, given a source database I over schema S, a target database J over schema T such that (I,J) satisfies Σ. In this case J is called a solution for the data exchange problem for I and Σ. A solution J is universal, if there is a homomorphism from J to any other solution J' of the setting. In RDF terms such a homomorphisms is the identity function on IRIs and literals, but allows blank nodes to be mapped either to other blank nodes or to IRIs or literals.

Different classes of constraints have been considered. An important class are tupel generating dependencies (TGD). These are roughly SPARQLog rules where the quantifier alternation is restricted to one alternation of the form $\forall\exists$. Therefore a data exchange problem for TGDs resembles the problem of computing the semantics for a given SPARQLog program P with dataset D. Hence the set of universal solutions could be considered a suitable semantics for P. Universal solutions have a drawback though: they are not closed under homomorphism: Given a universal solution J for a set of constraints Σ, there may be databases J' such that there is a homomorphism from J to J', yet J' is not even a solution (let alone a universal solution) for Σ. For instance, let $\Sigma = \{r(a,b), \forall x,y\exists z : r(x,y) \rightarrow s(x,z), \forall x : s(x,x) \rightarrow q(x)\}$. Then $J = \{r(a,b), \exists x : s(a,x)\}$ is a universal solution for Σ and there is a homomorphism from J to $K = \{r(a,b), s(a,a)\}$ (the identity function up to x which is mapped to a). However, K is not a solution of Σ (and thus also no universal solution) as $s(a,a)$ requires also $q(a)$ to hold in a solution for Σ due to the third constraint. Still it is easy to see that the set of universal solutions of P is a subset of the denotational semantics $[\![P]\!]$ defined in Section 1.4.2. In addition we show that the operational semantics [P] is a universal solution (Lemma 1.2). This is not surprising since our operational semantics is closely related to the chase procedure which can be used to compute universal solutions [4]. Nonetheless one could state that we have extended the chase procedure to a wider class of constraints (by allowing arbitrary quantifier alternation) and to a more general data model where the input database can contain blank nodes.

It has been observed in [8] that TGDs are not closed under composition. That is there are two finite sets of TGDs Σ_1 and Σ_2 such that there is no finite set of TGDs that defines the same database to database mapping as $\Sigma_1 \circ \Sigma_2$. Nonetheless [8] shows that there is a finite set Σ of non-TGDs constraints that is equivalent to $\Sigma_1 \circ \Sigma_2$. It turns out that this set Σ is in fact a set constraints with $\forall\exists\forall$ quantifier alternation that can be expressed in SPARQLog. This shows that if projection is not allowed (as in the standard data exchange setting or when all rules in a SPARQLog program are required to construct into the default graph), then the extra quantifier alternation does indeed add expressive power.

1.4 SPARQLog: SPARQL with Rules and Quantification

SPARQL has quickly become *the* standard for querying RDF data. Part of its success is certainly that it is a fairly compact language. Here we want to investigate how SPARQL can be extended with two features, rules and arbitrary quantification, without sacrificing most of its simplicity and the basic flavor of the language.

Rules are an acknowledged part of the Semantic Web vision. The `CONSTRUCT` query form of SPARQL can be seen as a form of non-recursive single-rule program and offer an obvious start point for a full rule language. With full rules SPARQL becomes by itself capable of computing such important concepts as the subsumption hierarchy in RDFS data or a person's social network in FOAF (*friend of a friend*) data.

The following SPARQLog program illustrates how we extend SPARQL's syntax to accommodate rules: Rather than a single `CONSTRUCT-WHERE` clause, in the following called *rule*, we allow any number of these to occur in the document. The bodies of SPARQLog rules are mostly standard SPARQL `WHERE` clauses. The heads are `CONSTRUCT` clauses that may be adorned with `GRAPH` patterns that specify the target graph of the rule. If no such pattern is present, the result of the rule is added to the default graph of the dataset. If such a pattern is present, it is added to the graph named in the `GRAPH` pattern. This allows, e.g., for hiding intermediary results. Note, that the dataset clauses `FROM` and `FROM NAMED` occur only once for all rules. Thus all rules query the same dataset.

```
PREFIX    :     <http://example.org/#ns>.
PREFIX    foaf: <http://xmlns.com/foaf/0.1/>.
PREFIX    wine: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
FROM       <http://example.org>
FROM NAMED <http://example.org/bavarians>

ALL ?x ?y
CONSTRUCT { ?x rdf:type :bavarian-look-alike }                              R3
WHERE { { { ?x :likes     ?y .          ?y rdf:type     wine:Wine .
            ?y wine:locatedIn wine:ItalianRegion } UNION
          { ?x :likes     ?y .          ?y rdf:type     wine:Wine .
            ?y wine:locatedIn wine:FrenchRegion } } .
        GRAPH <http://example.org/bavarians>
          { ?x rdf:type   :bavarian }

ALL ?x
CONSTRUCT GRAPH <http://example.org/bavarians>                              R2
          { ?x rdf:type   :bavarian }
WHERE     { ?x rdf:type   :european.    ?x foaf:knows   "Edmund" }

ALL ?x
CONSTRUCT { ?x rdf:type   :european }                                       R1
WHERE     { ?x foaf:knows "Angela" .    ?x foaf:knows   "Nicolas" .
            ?x foaf:knows "Elisabeth" }
```

In the example above, R_1 queries the default dataset (there is no `GRAPH` pattern in the body) to find people who know the German chancellor, the French president, and the British queen. Triples classifying these people as European are added to the default graph. In R_2 that default graph is queried for such people that also know the former Bavarian prime minister. Triples classifying such people as Bavarian are added to the named graph http://example.org/bavarians that is also included in the

dataset in the dataset clause of line 5. R_3 queries that named graph (containing the results of R_2 as well as any statements contained in the graph from the beginning) for Bavarians. If the default graph contains the information that such a person also likes Italian or French wine, then we that person can not be a real Bavarian, but must be a look-alike posing as a Bavarian.

Note that all rules carry explicit quantification clauses, e.g., `ALL ?x`. These clauses are added to support SPARQLog's second major addition over SPARQL:

SPARQL's `CONSTRUCT` rules allow the construction of RDF graphs. Unfortunately, the construction of one of RDF's most significant innovations, the provision of existential information in form of *blank nodes*, is poorly supported: Blank nodes can be constructed, but they are always scoped over all (universal) variables of the query. This makes most grouping tasks involving blank nodes (e.g., the construction of a container for all authors for each paper) impossible. In absence of rules this is a significant limitation. We suggest the use of an explicit quantifier clause to overcome this limitation: Rather than assuming that all blank nodes to be constructed are in the scope of all universal variables, we allow arbitrary quantifier quantification as in first-order logic.

To illustrate the issue of quantifier alternation and what kind of queries can be expressed, consider three examples. We choose to use explicit existential variables rather than blank nodes as their use in SPARQL is somewhat confusing (in bodies they play the role of normal variables, but in heads they are treated as existential).

The first, B_1, asserts that there is a Presenter for each TalkEvent who also attends that same event.

```
PREFIX eswc: <http://www.eswc2006.org/technologies/ontology#>

ALL ?x EX ?y                                                              B1
CONSTRUCT { ?y eswc:attendeeAt ?x . ?y rdf:type eswc:Presenter }
WHERE     { ?x rdf:type        eswc:TalkEvent }
```

The second, B_2, asserts that there is a MeetingRoomPlace that is the location of all talks.

```
EX ?x ALL ?y                                                              B2
CONSTRUCT { ?y eswc:hasLocation ?x . ?x rdf:type eswc:MeetingRoomPlace }
WHERE     { ?y rdf:type         eswc:TalkEvent }
```

The third, B_3, asserts that there for each TalkEvent there is someone that holds that talk and therefore is known by all attendees of the talk.

```
ALL ?x EX ?y ALL ?z                                                       B3
CONSTRUCT { ?x eswc:heldBy ?y .            ?z foaf:knows ?y }
WHERE     { ?x rdf:type    eswc:TalkEvent. ?z eswc:attendeeAt ?x }
```

The difference between the three cases is, of course, the scope of the existential variable (or blank node): In the first case, which is the only one unmodified SPARQL supports, one *fresh* blank node is created for each binding of the universal variable. In the second case, a single blank node is created that is the object in all hasLocation triples. In the third case, one *fresh* blank node is created for each binding of `?x` (each talk), but that same blank node is object in all knows triples for attendees of that talk.

It is worth noting that the latter two cases are essential for constructing many group or set like structures in RDF. For instance for RDF containers, RDF representations of *n*-ary relations [16], and RDF reification it is best and common practice to use blank nodes for the container, relation, or statement object. Unfortunately, SPARQL does not support either of these cases.

In presence of rules (and thus not in SPARQL) and graph projection we show in Section 1.5.2 that quantifier alternation, though convenient, actually does not increase the expressiveness. In other words, if rules and graph projection is present cases 2 and 3 can be expressed using only rules as in case 1. Nevertheless, even in these cases quantifier alternation is far more convenient (see rewriting in Section 1.5.2).

1.4.1 SPARQLog Syntax

As illustrated above, SPARQLog mostly adds rules and quantifier alternation to SPARQL. There are a few other restrictions and simplifications:

1. We do not consider `OPTIONAL` and `FILTER` expressions in this chapter. As negation in SPARQL is expressed via `OPTIONAL` and `isBound`, SPARQLog as presented here is based on positive SPARQL.
2. As stated above, we do not allow blank nodes to occur in bodies or heads of SPARQLog rules. In bodies blank nodes can be replaced by fresh universal variables, in heads with fresh existential ones.
3. We allow only IRIs or universal variables, but no existential variables, for the graph identifier (directly after `GRAPH`) in rule heads.
4. In contrast to SPARQL, we do require that all SPARQLog rules are range-restricted: If *x* is an universal variable in the head of a rule *R* than *x* must occur in the body of *R*. If there is an existential variable *y* in the head of *R* that is in the scope of an universal variable *x*, then *x* must occur in the body of *R*.
5. For simplicity, we only consider basic triple patterns and none of the abbreviation syntaxes from SPARQL (no predicate-object or object lists, no syntactic sugar for collections or `rdf:type`).

Literals and IRIs are as in RDF graphs (see Section 1.2), variables are from an infinite set disjoint with IRIs, literals, and blank nodes.

With the above restrictions, Figure 1.1 gives the syntax of SPARQLog. For simplicity, we ignore namespace prefixes and all associated issues in the following. Prefix resolution can be added easily, but only distracts from the salient points of this chapter. We allow `WHERE` clauses to be omitted entirely. In this case the rule head is always true (a fact).

⟨*program*⟩ ::= ⟨*prefix-clause*⟩* ⟨*dataset-clause*⟩ ⟨*rule*⟩+
⟨*dataset-clause*⟩ ::= 'FROM' ⟨*iri*⟩ ('FROM' 'NAMED' ⟨*iri*⟩)*
⟨*prefix-clause*⟩ ::= 'PREFIX' ⟨*identifier*⟩? ':' ⟨*iri*⟩
⟨*rule*⟩ ::= ⟨*quantifier-clause*⟩* ⟨*construct-clause*⟩ ⟨*where-clause*⟩?
⟨*quantifier-clause*⟩ ::= 'ALL' ⟨*variable*⟩+ | 'EX' ⟨*variable*⟩+
⟨*construct-clause*⟩ ::= 'CONSTRUCT' '{' ⟨*triple-pattern*⟩ '}'? ⟨*construct-template*⟩*
⟨*construct-template*⟩ ::= 'GRAPH' (⟨*iri*⟩ | ⟨*variable*⟩) '{' ⟨*triple-pattern*⟩ '}'
⟨*where-clause*⟩ ::= 'WHERE' '{' ⟨*graph-pattern*⟩ '}'
⟨*graph-pattern*⟩ ::= ('GRAPH' (⟨*iri*⟩ | ⟨*variable*⟩))? '{' ⟨*graph-pattern*⟩ '}'
| '{'⟨*graph-pattern*⟩'}' 'UNION' '{'⟨*graph-pattern*⟩'}'
| ⟨*graph-pattern*⟩ ('.' ⟨*graph-pattern*⟩)?
⟨*basic-graph-pattern*⟩ ::= ⟨*triple-pattern*⟩ ('.' ⟨*basic-graph-pattern*⟩)?
⟨*triple-pattern*⟩ ::= ⟨*resource*⟩ ⟨*predicate*⟩ ⟨*resource*⟩
⟨*resource*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩ | ⟨*literal*⟩
⟨*predicate*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩
⟨*variable*⟩ ::= '?' ⟨*identifier*⟩

Fig. 1.1 Syntax of SPARQLog

1.4.2 Denotational Semantics for SPARQLog

SPARQL's semantic is defined with a rather ad-hoc algebra in [19]. A more complete algebraic semantics of SPARQL is given in Chapter 9 of this volume. For SPARQLog a semantics based on (simple) RDF entailment seems the most natural, in particular since RDF entailment coincides with FO entailment of the canonical formulas of RDF datasets.

To this end, we first define a canonical formula also for SPARQLog programs. This can be seen as a translation of SPARQLog to first-order logic.

Definition 1.7 (Canonical Formula for SPARQLog Program). Let P be a SPARQLog program and $u_1,\ldots,u_n$ the identifiers for RDF graphs $G_1,\ldots,G_n$. Then the canonical formula $\phi(P)$ of P is defined as follows:

$\phi(\texttt{FROM}\ u_i\ Q)$	$= \bigwedge_{(s,p,o)\in G_i}(s,p,o,\diamond) \wedge \phi(Q)$
$\phi(\texttt{FROM NAMED}\ u_i\ Q)$	$= \bigwedge_{(s,p,o)\in G_i}(s,p,o,u_i) \wedge \phi(Q)$
$\phi(\mathit{ALL\ vars}\ Q)$	$= \forall\ \mathit{vars} : \phi(Q)$
$\phi(\mathit{EX\ vars}\ Q)$	$= \exists\ \mathit{vars} : \phi(Q)$
$\phi(\texttt{CONSTRUCT}\ \mathit{template}\ Q)$	$= \phi_g(\mathit{template},\diamond)\ \phi(Q)$
$\phi(\texttt{WHERE}\ \mathit{pattern})$	$= \leftarrow \phi_g(\mathit{pattern},\diamond)$
$\phi_g(\mathit{pattern}\ \texttt{UNION}\ Q,c)$	$= \phi_g(\mathit{pattern},c) \vee \phi_g(Q,c)$
$\phi_g(\texttt{GRAPH}\ \mathit{var}\ Q,c)$	$= \phi_g(Q,\mathit{var})$
$\phi_g(\texttt{GRAPH}\ \mathit{iri}\ Q,c)$	$= \phi_g(Q,\mathit{iri})$
$\phi_g(\mathit{pattern}\ \texttt{.}\ Q,c)$	$= \phi_g(\mathit{pattern},c) \wedge \phi_g(Q,c)$
$\phi_g(\texttt{\{}\ \mathit{pattern}\ \texttt{\}},c)$	$= \phi_g((\mathit{pattern}),c)$
$\phi_g(\mathit{sub\ pred\ obj},\mathrm{c})$	$= R(\phi(\mathit{sub}),\phi(\mathit{pred}),\phi(\mathit{obj}),c)$

The canonical formula is pretty straightforward: the head of a SPARQLog rule, i.e., the `CONSTRUCT` clause, is translated into a conjunction that forms the consequence of an implication. The `WHERE` clause is translated into the condition of the implication. It may contain both disjunctions and conjunctions. Noteworthy is the propagation of the identifier of the current query graph in the dataset by means of the second parameter of ϕ_g.

For instance, rule B_3 from Section 1.4 has the following canonical formula (brackets are normalized):

$$\forall x \exists y \forall z : R(x, \mathsf{eswc{:}heldBy}, y, \diamond) \wedge R(z, \mathsf{foaf{:}knows}, y, \diamond) \\ \leftarrow R(x, \mathsf{rdf{:}type}, \mathsf{eswc{:}TalkEvent}, \diamond) \wedge R(z, eswc : attendeeAt, x, \diamond).$$

Here all triples occur in the default graph. In contrast, the canonical formula for rule R_2 highlights that the default graph is queried, but that the consequences are triples in a named graph:

$$\forall x : R(x, \mathsf{rdf{:}type}, \mathsf{:bavarian}, \texttt{<http://example.org/bavarians>}) \\ \leftarrow R(x, \mathsf{rdf{:}type}, \mathsf{:european}, \diamond) \wedge R(z, \mathsf{foaf{:}knows}, \text{"Edmund"}, \diamond).$$

It is not generally agreed upon what the semantics of a rule based RDF query language should be if existential variables are allowed in the head. In contrast, it is agreed that the semantics of a logic program with only universally quantified variables is its minimal Herbrand model.

We deal with this problem by defining the semantics of SPARQLog in terms of entailment. More precisely we define the semantics of an SPARQLog program P to be the set of all RDF datasets D whose canonical formulas entail the same RDF datasets as the canonical formula of P.

Definition 1.8 (Denotational Semantics of SPARQLog). Let P be an SPARQLog program. The *denotational semantics* $[\![P]\!]$ of P is the set of all RDF datasets D, such that for all RDF datasets E it holds that $\phi(\mathsf{D})$ entails $\phi(\mathsf{E})$ if and only if $\phi(P)$ entails $\phi(\mathsf{E})$.

Observe that the semantics of an SPARQLog program is an infinite set of possibly infinite RDF datasets. However, we choose the above semantics (and not, e.g., the set of all datasets whose canonical formulas follow logically from $\phi(P)$) to ensure that $[\![P]\!]$ forms an equivalence class under (RDF) entailment. Therefore any element (in particular, any finite element if such exists) of $[\![P]\!]$ characterizes the entire set. We consider an implementation of SPARQLog *sound* and *complete* if it returns any element of $[\![P]\!]$ for a given SPARQLog program P.

This semantics is not RDFS aware. However, as stated above [15] gives a set of first-order formulas that characterize the logical core of RDFS (there called ρfs). These formulas can be expressed by SPARQLog rules and added to a program, if an RDFS aware semantics is desired.

1.4.3 Relational Operational Semantics for SPARLog

The goal of this section is to give an evaluation of an SPARQLog program P by first translating P into a logic program $s(P)$, using the well-studied notion of Skolemisation [5], and then evaluating this program $s(P)$ using standard logic programming or relational technology. Two post processing steps (Unskolemisation and RDF normalization) make sure that the result is an RDF graph in the denotational semantics of P.

We use the well studied notion of Skolemisation [5] to translate an SPARQLog program into a logic program:

Definition 1.9 (Skolemisation [5]). Let Σ and Γ be disjoint alphabets, $\varphi = \forall \bar{x} \exists y(\psi)$ a formula over $\Sigma \cup \Gamma$ and $f \in \Gamma$. A Γ-*Skolemisation step* s_f maps φ to $s_f(\varphi) := \forall \bar{x} \psi\{y \leftarrow f(\bar{x})\}$. (We denote with $\phi\{t \leftarrow t'\}$ the formula ϕ where all occurrences of the term t are replaced by the term t'). A Γ-*Skolemisation* s is a composition $s_{f_1} \circ \ldots \circ s_{f_n}$ of Γ-Skolemisation steps such that f_i does not occur in $s_{f_{i+1}} \circ \ldots \circ s_{f_n}(\varphi)$ and $s(\varphi)$ contains no existential variables. The definition of a Skolemisation is extended to sets in the usual way.

The Skolemisation of (the canonical formula of) an SPARQLog program P is equivalent to a range restricted logic program, which we denote by $s(\phi(P))$. If necessary, disjunction in rule bodies and conjunction in rule heads is expanded into multiple rules as usual. Any logic programming engine can compute the minimal Herbrand model $M_{s(\phi(P))}$ of $s(\phi(P))$.

For instance, the following logic program is the Skolemisation $s(\phi(P))$ of the SPARQLog rule B_3 from Section 1.4 where s replaces the existential variable y in P by the term $s_y(x)$:

$$
\begin{aligned}
\big\{ & \forall xz : R(x, \mathsf{eswc{:}heldBy}, s_y(x), \diamond) \\
& \quad \leftarrow R(x, \mathsf{rdf{:}type}, \mathsf{eswc{:}TalkEvent}, \diamond) \wedge R(z, eswc : attendeeAt, x, \diamond). \\
& \forall xz : R(z, \mathsf{foaf{:}knows}, s_y(x), \diamond) \\
& \quad \leftarrow R(x, \mathsf{rdf{:}type}, \mathsf{eswc{:}TalkEvent}, \diamond) \wedge R(z, eswc : attendeeAt, x, \diamond). \big\}
\end{aligned}
$$

We define $\phi(M_{s(\phi(P))})$ to be the conjunction of all ground atoms that are true in $M_{s(\phi(P))}$. However, $\phi(M_{S(P)})$ might not be the canonical formula of an element of $[\![P]\!]$ for two reasons. First, the example shows that $\phi(M_{S(P)})$ might contain atoms with skolem terms, which are not entailed by $\phi(P)$. Second, $\phi(M_{S(P)})$ can contain atoms with literals in subject or predicate position or blank nodes in predicate position. Such atoms are not allowed in an RDF graph and therefore never part of an element of $[\![P]\!]$.

We can avoid the first problem by "undoing" the Skolemisation: replacing each Skolem term in $\phi(M_{S(P)})$ by a fresh, distinct blank node. We formalise this operation as the inverse of a Skolemisation called *Unskolemisation*.

Definition 1.10 (Unskolemisation). Let Σ and Γ be disjoint alphabets and φ a ground, possibly infinite, and quantifier free formula over $\Sigma \cup \Gamma$. Let $\bar{t}$ be the sequence of all ground terms $f(\bar{u})$ where f is in Γ and $\bar{u}$ is a sequence of terms over $\Sigma \cup \Gamma$. Then the *Γ-Unskolemisation u* maps φ to $u(\varphi) := \exists \bar{x}(\varphi\{\bar{t} \leftarrow \bar{x}\})$. where $\bar{x}$ is a sequence of fresh variables.

To address the second issue, we remove all atoms with literals or blank nodes in predicate position (no RDF graph may contain such a triple or any triple entailed by it). In addition we remove each triple t in graph u that contains a literal l in subject position and add two triples t_1 and t_2 to u where t_1 is obtained from t by replacing an occurrence of a literal l in subject position by a fresh blank node b_l and t_2 is obtained from t by replacing all occurrences of l by b_l. Dropping these atoms does not affect the soundness or completeness (see below) as these atoms can by definition not be part of an RDF dataset. Nevertheless, allowing them during the evaluation is useful and even necessary for certain programs (for details see [15]).

Definition 1.11 (Normalisation Operator). Let φ be a formula of the form $\exists \bar{x}(a_1(\bar{x}) \wedge \ldots \wedge a_n(\bar{x}))$ where each $a_i(\bar{x}) = T(t_1,t_2,t_3)$ for some $t_1,t_2,t_3 \in (\mathsf{U} \cup \mathsf{B} \cup \mathsf{L})$. Let $\mathsf{L}' \subseteq \mathsf{L}$ be the set of literals that occur in the first argument of an atom in φ. We define $\mu : \mathsf{U} \cup \mathsf{B} \cup \mathsf{L} \to \mathsf{U} \cup \mathsf{B} \cup \mathsf{L}$ to be the injection such that $\mu(t) = b$ for some fresh blank node b (not in φ) if $t \in \mathsf{L}'$ and $\mu(t) = t$ otherwise. Then $\Pi(\varphi) = \{\Pi(a_1(\bar{x})), \ldots \Pi(a_n(\bar{x}))\}$ and

$$\Pi(T(t_1,t_2,t_3)) = \begin{cases} \top & \text{if } t_2 \in \mathsf{B} \cup \mathsf{L} \\ (\mu(t_1),t_2,t_3) \wedge (\mu(t_1),t_2,\mu(t_3)) & \text{otherwise} \end{cases}$$

The normalisation operator ensures that, though intermediary atoms may contain blank nodes in predicate position (see [24] for examples where this is useful), the final answer of an SPARQLog program never contains such atoms.

Armed with these notions of Skolemisation, Unskolemisation and Normalisation, we finally define the operational semantics of SPARQLog as follows:

Definition 1.12 (Operational Semantics of SPARQLog). Let P be an SPARQLog program over Σ, s a Γ-Skolemisation for P, and u an Γ-Unskolemisation. Then the *operational semantics* $[P]$ of P is $[P] := \Pi\left(u(\phi(M_{s(\phi(P))}))\right)$ where $\phi(M_{s(\phi(P))})$ is as defined above: the conjunction of all ground atoms that are true in the minimal Herbrand model of $s(\phi(P))$.

1.4.3.1 Soundness and Completeness

Even though we do not require that elements of the denotational semantics $[\![P]\!]$ of an SPARQLog program P are models of P it holds that $u(\phi(M_{s(\phi(P))}))$ has a canonical structure that is not only a model of P but even a universal model (or universal solution in the sense of [7]). Thus if we allow literals in subject position and blank nodes in subject or predicate position, we can omit Π from the operational semantics and compute a model of P.

To formulate this more precisely, we define an *extended Herbrand structure A* over alphabet Σ and variables *Var* as a structure (D, Rel, Fun) where D is the set of (possibly non-ground) terms over Σ and *Var*, and every function f^A is defined by $f^A(t_1,\ldots,t_n) = f(t_1,\ldots,t_n)$. We extend the definition of Unskolemisation from formulas to extended Herbrand structures: if u is an Unskolemisation that replaces $\bar{t}$ by $\bar{x}$ then $u(M)$ is the extended Herbrand structure obtained from M by renaming the domain elements $\bar{t}$ by $\bar{x}$.

Lemma 1.2. *Let P be an SPARQLog program. Then* $A_P = u(M_{s(\phi(P))}) \models \phi(P)$ *and* $\phi(P) \models u\left(\phi(M_{s(\phi(P))})\right) = \psi_P$.

Intuitively, $A_P \models \phi(P)$ means that ψ_P captures all the information in P and $\phi(P) \models \psi_P$ means that it does not assert anything that is not asserted by P. From these two key observations, we can prove that the operational semantics of SPARQLog is both sound and complete with respect to the denotational semantics.

Theorem 1.1. *Let P be an SPARQLog program. Then* $[P] \in [\![P]\!]$.

1.4.3.2 Proof of Lemma 1.2 and Theorem 1.1

In the proof of Lemma 1.2 we often make use of the Substitution lemma, which we state here without proof.

Lemma 1.3 (Substitution Lemma). *Let φ be a sentence and M an interpretation. Then*

$$M \models \varphi \quad \textit{iff} \quad M, d \models \psi(x)$$

if $\psi = \varphi\{t \leftarrow x\}$ *and M interpretes t as d.*

We now recall some well known results about Skolemisation. Symmetric proofs show the following properties of Unskolemisation.

Lemma 1.4 (Skolemisation Lemma). *Let Σ, Γ and Π be disjoint alphabets and φ a finite formula over $\Sigma \cup \Gamma$. Let s be a Π-Skolemisation for φ, u an Γ-Unskolemisation for φ. Then*

- $\varphi \models u(\varphi)$
- $s(\varphi) \models \varphi$.
- $u(\varphi)$ *is satisfiable iff* φ *is satisfiable.*
- φ *is satisfiable iff* $s(\varphi)$ *is satisfiable.*

Corollary 1.1 (of the Skolemisation Lemma). *Let φ be a finite formula over $\Sigma \cup \Gamma$ and u an Γ-Unskolemisation for φ. If S is a model of $u(\varphi)$ over Σ then there exists an extension T of S on Γ which is a model of φ.*

Before we can turn to the actual proofs of the soundness and completeness of the operational semantics, we have to establish some further properties of Unskolemization. These are the central observations to show that every RDF dataset which is entailed by the operational semantics of an SPARQLog program P is entailed by $\phi(P)$.

Lemma 1.5. *Let φ and ψ be formulas over $\Sigma \cup \Gamma$ where φ is finite and ψ is possibly infinite and ground. Let u be an Γ-Unskolemisation for φ. Then*

$$\varphi \models \psi \quad \textit{implies that} \quad u(\varphi) \models u(\psi).$$

Proof. Let S be a model over Σ of $u(\varphi)$. As φ is finite, by Corollary 1.1 there is an extension T of S on Γ which is a model of φ. By the assumption T is also a model of ψ. Then it follows from Lemma 1.4 that T is a model of $u(\psi)$. As $u(\psi)$ contains no symbol from Γ and T is an extension of S on Γ, S is a model of $u(\psi)$.

Lemma 1.6. *Let φ be a formula over Σ, M an extended Herbrand structure over Σ and Var, and u an Γ-Unskolemisation for φ. Then*

$$M \models \varphi \quad \textit{implies that} \quad u(M) \models u(\varphi)$$

Proof. Assume that $M \models \varphi$. Let $\forall \bar{x}(\psi) = \varphi$. Then for all sequences of terms $\bar{t}$ it holds that $M \models \psi(\bar{t})$. As M is an extended Herbrand structure, it interprets every constant c by c. Therefore it follows from the substitution Lemma that $M \models (\psi\{\bar{c} \leftarrow \bar{y}\})(\bar{t},$
barc). Observe that $u(M)$ be the extended Herbrand structure obtained from M by renaming the domain elements $\bar{c}$ by $\bar{y}$. Thus $u(M) \models (\psi\{\bar{c} \leftarrow \bar{y}\})(\bar{t})$. Finally by the definition of entailment and Unskolemisation it holds that $u(M) \models u(\varphi)$.

Proof of Lemma 1.2. Let P be an SPARQLog program over alphabet Σ. We need to show that $A_P = u(M_{s(\phi(P))}) \models \phi(P)$ and $\phi(P) \models u\left(\phi(M_{s(\phi(P))})\right) = \psi_P$.

To show that $A_P \models \phi(P)$, observe that by definition $M_{s(\phi(P))}$ is a model of $s(\phi(P))$. It therefore follows from Lemma 1.6 that $u(M_{s(\phi(P))})$ is a model of P.

For the second part observe that as $s(\phi(P))$ is a logic program it follows that $s(\phi(P))$ entails each atom that is true in $M_{s(\phi(P))}$. Thus $s(\phi(P))$ also entails the canonical formula $\phi(M_{s(\phi(P))})$ of $M_{s(\phi(P))}$. Let u be the inverse of s. As $s(\phi(P))$ is a finite set of finite formulas and ψ_P is a ground formula it follows from Lemma 1.5 that $u \circ s(\phi(P)) = \phi(P)$ entails $u\left(\phi(M_{s(\phi(P))})\right)$.

Proof of Theorem 1.1. The aim is to show that every RDF dataset that is entailed by an SPARQLog program is also entailed by the operational semantics. First we need to establish a few more properties of canonical formulas of SPARQLog programs:

Lemma 1.7. *Let P be a logic program over alphabet Σ and M_P its minimal Herbrand model. Let* g *be an RDF dataset and $\phi(\mathrm{g}) = \exists \bar{x}(\bigwedge \psi)$ its canonical formula. Then the following statements are equivalent*

(a) $P \models \bigwedge \psi\{\bar{x} \leftarrow \bar{t}\}$ for some sequence of variables $\bar{x}$ and ground terms $\bar{t}$
(b) $P \models \phi(\mathrm{g})$
(c) $M_P \models \phi(\mathrm{g})$

Proof. It is trivial that (a) implies (b). To see that (b) implies (c) observe that M_P is a model of P. To show that (c) implies (a) assume that (c) is true. As M_P is a Herbrand model there is a sequence of terms $\bar{t}$ such that $M_P, \bar{t}$ is a model of $\bigwedge \Phi(\bar{x})$. In addition,

M_P interprets all terms by themselves. Thus it follows from the substitution lemma that M_P is a model of $\bigwedge \Phi\{\bar{x} \leftarrow \bar{t}\}$. Therefore M_P is a model of $a\{\bar{x} \leftarrow \bar{t}\}$ for every $a \in \Phi$. As $a\{\bar{x} \leftarrow \bar{t}\}$ is a ground atom it follows that $P \models a\{\bar{x} \leftarrow \bar{t}\}$. As this is true for every $a \in \Phi$ it holds that $P \models \bigwedge \Phi\{\bar{x} \leftarrow \bar{t}\}$ for some sequence of terms $\bar{t}$.

With these properties we can now show the Theorem 1.1: Let P be an SPARQLog program and $\psi_P = u(\phi(M_{s(\phi(P))}))$. We first show that that for any RDF graph g

$$\phi(P) \models \phi(\mathrm{g}) \quad \text{iff} \quad \psi_P \models \phi(\mathrm{g}).$$

The direction from right to left follows from the second part of Theorem 1.2. For the direction from left to right let $\phi(\mathrm{g}) = \exists \bar{x}(\bigwedge \xi)$ where ξ is a set of atom. Assume that $\phi(P) \models \phi(\mathrm{g})$. By Lemma 1.4 $s(\phi(P)) \models \phi(\mathrm{g})$ for any Skolemisation s of $\phi(P)$. As $s(\phi(P))$ is a logic program it follows from Lemma 1.7 that there is a sequence $\bar{t}$ of terms such that $s(\phi(P)) \models \bigwedge \xi\{\bar{x} \leftarrow \bar{t}\}$. Thus for all atoms $a \in \xi\{\bar{x} \leftarrow \bar{t}\}$ it holds that $s(\phi(P)) \models a$. As $M_{s(\phi(P))}$ is a model of $s(\phi(P))$ it follows that $M_{s(\phi(P))}$ is a model of a. Let $\bigwedge M_{s(\phi(P))}$ be the conjunction of all ground atoms which are true in $M_{s(\phi(P))}$. Then a is a conjunct in $M_{s(\phi(P))}$ and thus $\bigwedge M_{s(\phi(P))} \models a$. As this is true for any $a \in \xi\{\bar{x} \leftarrow \bar{t}\}$ it holds that $\bigwedge M_{s(\phi(P))} \models \xi\{\bar{x} \leftarrow \bar{t}\}$.

Thus $\bigwedge M_{s(\phi(P))} \models \phi(\mathrm{g})$ and there is a homomorphism μ from $M_{s(\phi(P))}$ to g. Observe that there is a mapping ν from $D^{M_{s(\phi(P))}}$ to $D^{u(M_{s(\phi(P))})}$ such that (i) $\nu(c^{M_{s(\phi(P))}}) = c^{u(M_{s(\phi(P))})}$ if c is a IRI or literal, (ii) if f is a skolem symbol then $\nu(f(\bar{t})) = x_{f(\bar{t})}$ where $x_{f(\bar{t})}$ is a non-constant domain element in $D^{u(M_{s(\phi(P))})}$, and (iii) $R^{M_{s(\phi(P))}}(\bar{d})$ iff $R^{u(M_{s(\phi(P))})}(\nu(\bar{t}))$ for every relation symbol R. Observe that $\mu \circ \nu$ is a homomorphism from G to $u(M_{s(\phi(P))})$. Thus the operational semantics $u(M_{s(\phi(P))})$ of P entails φ^{g}.

It remains to show that $\theta \models \phi(\mathrm{g})$ iff $\Pi(\theta) \models \phi(\mathrm{g})$ where θ is a formula as in the definition of the normalisation operator Π. The direction from right to left is immediate since $\Pi(\theta) \models \theta$. The other direction follows from the definition of Π and the structure of RDF triples.

1.5 Properties of SPARQLog

1.5.1 Designing Tractable Fragments of SPARQLog

Since the full SPARQLog captures some classes of expressive formulas (such as the $\forall\exists$-rules) it is easy to adapt some standard proofs of Turing completeness (see, e.g.,[4]) to show the following.

Proposition 1.1. *SPARQLog is Turing complete.*

This section focuses on the description of fragments of SPARQLog, recognizable in PTIME, that ensure polynomial complexity when given a fixed (or fairly small)

program P and a potentially very large RDF dataset (playing the role of a *database*). We can formalize the desired notion of tractability as follows:

Definition 1.13 (Tractability). We say that an SPARQLog program P containing the RDF graph identifiers $u_1,\ldots,u_n$ in its dataset clause is tractable iff the following holds: For all RDF graphs $G_1,\ldots,G_n$ associated with $u_1,\ldots,u_n$ of total size n, the RDF dataset $[P]$ is finite and can be computed in time $O(n^k)$ for some k depending only on P.

It follows from Theorem 2 in [13] that the finiteness of $[P]$ (for all $G_1,\ldots,G_n$) is actually a sufficient condition for polynomial data-complexity. By using a standard encoding of general relational constraints into RDF constraints, we can adapt the proof of Theorem 4 in [13] to show that the finiteness of $[P]$ is undecidable.

Proposition 1.2. *The following problem is undecidable: given an SPARQLog program P, is P tractable?*

Note that the union $P_1 \cup P_2$ of two tractable SPARQLog programs P_1 and P_2 is not necessarily tractable. Consider for instance the two following rules, where `:a` and `:b` denotes two distinct IRIs:

```
R1 = ALL ?x ?y EX ?z
   CONSTRUCT { ?y <b> ?z } FROM { ?x <a> ?y }
R1 = ALL ?x ?y EX ?z
   CONSTRUCT { ?y <a> ?z } FROM { ?x <b> ?y }
```

Even though $P_1 = \{R_1\}$ and $P_2 = \{R_2\}$ are tractable, we can check that the SPARQLog program $P_{12} = \{R_1, R_2\}$ is *not* tractable. In particular, in the case of an RDF graph G containing a triple (`<c>`,`<a>`,`<d>`) we can observe that $[P]$ is infinite as it must contain an infinite path of the form

$$\{(\texttt{<c>},\texttt{<a>},\texttt{<d>});(\texttt{<d>},\texttt{<b>},\texttt{\_:1});(\texttt{\_:1},\texttt{<a>},\texttt{\_:2});(\texttt{\_:2},\texttt{<b>},\texttt{\_:3});\ldots\}$$

where `_:`i is a blank node.

As also illustrated by this example, there is very little hope of identifying an interesting *local* criterion (testing each rule independently) ensuring the right notion of tractability. In particular, the notion of *guarded Datalog*$^{\pm}$ from Chapter 11 of this volume, designed to ensure (only) the tractability of query answering, does not ensure the tractably of the data-exchange problem (i.e., the materialization of $[P]$). We can indeed observe that P_{12} is not tractable in the sense of Definition 1.13 even though it is *guarded* (each rule contains an atom in its body that contains all universally quantified variables of that rule).

A more relevant approach would consist in relying on the notion of *weak-acylicity* (WA), introduced in [6], and based on the study of two different processes: the creation of new terms in some positions and the migration of newly-created terms from initial positions to new positions. A criteria of acyclicity then ensures that there is no infinite loop in this process of creation and migration of new terms, and that the evaluation of $[P]$ terminates in polynomial time.

Even though WA is a fairly simple way of ensuring tractability, we argue in this section that the more technical notion of *Super-weak Acyclicity* (SwA) introduced in

[13] turns out to be a very significant and useful generalization of WA in the context of RDF.

First, SwA allows to take in account the numerous constants that usually occur in a SPARQLog program while relying on efficient unification technics to distinguish distinct constants. This contrasts with WA which was only defined for constraints without constants.

Second, SwA relies on a richer notion of *positions* (called *places*). A standard approach in the context of relational databases is indeed to define a *position* as a pair (R,A) where R is a relational symbol, and A is a single attribute (or column) of R. In the context of RDF, since we have only a single predicate symbol of small arity we would only consider a fixed and very small number (typically 3) of positions and large programs are almost never acyclic in the sense of WA. In contrast, the SwA relies on *places* of the form (a,i) where a is an atom of the logic program $s(\phi(P))$ and $i \in \{1,2,3\}$. SwA distinguishes therefore a polynomial number $O(||P||^3)$ of *places* instead of distinguishing only 3 positions.

Third, SwA enjoys some natural closure properties (missing in WA) which make the design of SwA programs easier (see Theorem 5 in [13]). In particular, adding more atoms in the body of some rule in P never hurts: if P is SwA then the resulting set of rules is also SwA.

Note that other generalizations of WA (incomparable with SwA) have been proposed in the literature, in particular, the notion of *Stratification* [4], the notion of *Safe Restriction* [22], and the notion of *Inductive Restriction* [14]. However, none of these notions solves the problems of WA discussed above. In particular, the tractable program

```
ALL ?x ?y EX ?z
CONSTRUCT { ?y <b> ?z } FROM { ?x <a> ?y }
```

belongs to none of these classes because none of them take into account the fact that the two IRIs (constants) a and b are distinct. Moreover, these three classes only ensure the termination of the so-called *restricted chase*, and – unlike SwA – they do not ensure the termination of the logic program $s(\phi(P))$ (i.e. $[P]$ could be infinite).

1.5.1.1 SwARQLog

We define in this section a tractable fragment called SwARQLog (for super-weakly acyclic SPARQLog) relying on the notion of *super-weak acyclicity* (SwA) introduced in [13] in the context of data-exchange with $\forall\exists$-rules only, and adapted here to the case of rules with quantifier alternation.

Given an SPARQLog program P we let P^* be the logic program $P^* = s(\phi(P))$ and define a *place* as a pair (a,i) where a is a atom of P^* and $i \leq 3$. We write $(a,i) \sim (a,i')$ when $i = i'$ and the atoms a and a' are unifiable. Given two sets of places Q and Q' we write $Q \sqsubseteq Q'$ iff for all $p \in Q$ there exists $p' \in Q'$ such that $p \sim p'$. Given a rule $r : B_r \rightarrow H_r$ and a variable x we let $\rho(x,B_r)$ and $\rho(x,H_r)$ be the set of places $(R(t_1,t_2,t_3,u),i)$ in the body B_r and the head H_r such that $t_i = x$. Given a function symbol f, we let $\rho(f,H_r)$ be the set of places $(R(t_1,t_2,t_3,u),i)$

$$\phi(P) = \begin{cases} \forall lec \exists crs \forall stu \ \big(R(lec, \text{rdf:type}, \text{uni:lecture}, \diamond) \wedge R(stu, \text{uni:attends}, lec, \diamond) \\ \qquad \rightarrow R(crs, \text{uni:practices}, lec, \diamond) \wedge R(stu, \text{uni:attends}, crs, \diamond) \big) \\ \forall x \exists prf \forall stu \ \big(R(stu, \text{uni:attends}, x) \\ \qquad \rightarrow R(x, \text{uni:taught-by}, prf, \diamond) \wedge R(prf, \text{people:knows}, stu, \diamond) \big) \end{cases}$$

$$s(\phi(P)) = \begin{cases} R(\underset{1}{lec}, \text{rdf:type}, \text{uni:lecture}, \diamond) \wedge R(\underset{2}{stu}, \text{uni:attends}, \underset{3}{lec}, \diamond) \\ \qquad \rightarrow R(\underset{4}{f(lec)}, \text{uni:practices}, \underset{5}{lec}, \diamond) \wedge R(\underset{6}{stu}, \text{uni:attends}, \underset{7}{f(lec)}, \diamond) \\ R(\underset{8}{stu}, \text{uni:attends}, \underset{9}{x}, \diamond) \\ \qquad \rightarrow R(\underset{10}{x}, \text{uni:taught-by}, \underset{11}{g(x)}, \diamond) \wedge R(\underset{12}{g(x)}, \text{people:knows}, \underset{13}{stu}, \diamond) \end{cases}$$

Fig. 1.2 Super-weakly acyclic program P

in the head H_r such that t_i is of the form $f(\ldots)$. Given a set of places Q we define $\mathsf{Fix}^*(Q)$ as the smallest set of places Q' such that $Q \subseteq Q'$ and for all rules $r : B_r \rightarrow H_r$ and all variables x we have $(\rho(x, B_r) \sqsubseteq Q') \Rightarrow (\rho(x, H_r) \subseteq Q')$. Let F^* be the set of function symbols occurring in P^*, then we can observe that each $f \in F^*$ occurs in exactly one rule denoted $r_f : B_f \rightarrow H_f$ and that each occurrence of f in this rule uses the same vector of arguments denoted $\mathsf{arg}(f)$. Given two function symbols $f \in F^*$ and $g \in F^*$ we say that f *feeds* g iff there exists some $x \in \mathsf{arg}(g)$ such that $\rho(x, B_g) \sqsubseteq \mathsf{Fix}^*(\rho(f, H_f))$ and we define the feeding graph $\mathscr{G}(P)$ of P as the graph $(F^*, \rightsquigarrow)$ containing an edge $(f \rightsquigarrow g)$ iff f feeds g.

Definition 1.14 (SwARQLog). An SPARQLog program P is super-weakly acyclic (SwA) and is called an SwARQLog program iff the feeding graph $\mathscr{G}(P)$ is acyclic.

Consider, for instance, the following SwARQLog program P:

```
PREFIX uni:  <http://example.org/uni>
FROM    <http://example.org/oxford>

ALL ?lec EX ?crs ALL ?stu
CONSTRUCT { ?crs uni:practices ?lec   . ?stu uni:attends ?crs }
WHERE     { ?lec rdf:type uni:lecture . ?stu uni:attends ?lec }

ALL ?lec EX ?prf ALL ?stu
CONSTRUCT { ?lec uni:taught-by ?prf   . ?prf foaf:knows  ?stu  }
WHERE     { ?stu uni:attends ?lec }
```

Figure 1.2 illustrates the places in P: $p_1 = \big(R(lec, \text{rdf:type}, \text{uni:lecture}, \diamond), 1\big)$; $p_2 = \big(R(stu, \text{uni:attends}, lec, \diamond), 1\big)$; $p_3 = \big(R(stu, \text{uni:attends}, lec, \diamond), 3\big)$; $\ldots$; $p_{13} = \big(R(prf, \text{people:knows}, stu, \diamond), 3\big)$. Note that there are more places, but we show only the useful ones. With letting f and g the skolem functions used in $s(\phi(P))$ we can check that P is indeed super-weakly acyclic:

- f feeds g: We have indeed $\rho(f, H_f) = \{p_4, p_7\}$ and $\mathsf{Fix}^*(\{p_4, p_7\}) = \{p_4, p_7, p_{10}\}$ while $\mathsf{arg}(g) = \{x\}$ and $\rho(x, B_g) = \{p_9\}$. Since p_9 unifies with p_{11} we have therefore $\rho(x, B_g) \sqsubseteq \mathsf{Fix}^*(\rho(f, H_f))$

- f does not feed f: We have indeed$\rho(f,H_f) = \{p_4,p_7\}$ and $\mathsf{Fix}^*(\{p_4,p_7\}) = \{p_4,p_7,p_{10}\}$ while $\mathsf{arg}(f) = \{lec\}$ and $\rho(lec,B_f) = \{p_1,p_3\}$. Since none of the places in $\{p_4,p_7,p_{10}\}$ unifies with p_1 we have $\rho(lec,B_f) \not\sqsubseteq \mathsf{Fix}^*(\rho(f,H_f))$.
- we can check similarly that g does not feed f and g does not feed g.

The definition above coincides precisely with the definition of SwA given in [13] for the case of $\forall\exists$-rules and we can easily adapt the proofs given in [13] to also cover quantifier alternation and thus show the following.

Theorem 1.2 (Tractability of SwARQLog).
(1) We can decide whether an SPARQLog program is SwA in PTIME.
(2) Every SwARQLog program is tractable.

1.5.2 Expressiveness of Quantifier Alternation in SPARQLog

SPARQLog allows existential variables in any position of the quantifier of a rule. This contrasts to other RDF query languages that are either limited to rules in $\forall\exists$ or to $\exists\forall$ form: In $\forall\exists$ approaches such as [21] existential quantifiers occur in the scope of *all* universal variables of a rule. In $\exists\forall$ approaches such as [25], existential variables occur in the scope of *no* universal variables.

In this section, we show that an SPARQLog program P can be translated into an SPARQLog program $F_{\forall\exists}(P)$ such that the two programs are default-graph equivalent and $F_{\forall\exists}(P)$ contains only rules in $\forall\exists$ form. Such an equivalence does not hold for the $\exists\forall$ form.

Default-graph equivalence captures the notion that they both construct the same default graph, but may differ on the named graphs they query and construct in intermediary rules:

Definition 1.15 (Default-graph Equivalence). Let P and P' be two SPARQLog programs. Then P is *default-graph equivalent* to P' if for all datasets $\mathsf{D} \in [\![P]\!], \mathsf{D}' \in [\![P']\!]$ it holds that $\mathsf{D}[\diamond] \models\!\models \mathsf{D}'[\diamond]$.

Thus two SPARQLog programs that are default-graph equivalent can be considered equivalent up to results in intermediary named graphs.

First, we define $F_{\forall\exists}$. For convenience, we abbreviate for any IRI H and sequence of variables $\bar{x} = x_1,\ldots,x_n$, the conjunction of triple patterns

```
(H rdf:_1 ?x_1) . \ldots . (H rdf:_n x_n)
```

by $H(\bar{x})$ and, for any graph IRI I,

$$\tilde{H}(\bar{x},I) = R(H,\texttt{rdf:\_1},x_1,I) \wedge \ldots \wedge R(H,\texttt{rdf:\_n},x_n,I).$$

Definition 1.16 ($\forall\exists$ Rewriting). Let P be an SPARQLog program and

```
1 ALL x̄ EX ȳ Qz̄
  CONSTRUCT { ξ(x̄,ȳ,z̄) } WHERE { ψ(x̄,ȳ,z̄) }
```

a rule R in P with $Q\bar{z}$ some sequence of quantifier clauses over the variables $\bar{z}$, $\xi(\bar{x},\bar{y},\bar{z})$ a construct template over the given variables and $\psi(\bar{x},\bar{y},\bar{z})$ a graph pattern over the given variables.

Then we define the $\forall\exists$-*rewriting* $F_{\forall\exists}(\phi)$ as

$$F_{\forall\exists}(\phi) = \begin{cases} R & \text{if } R \text{ is in } \forall\exists \text{ form} \\ R_1R_2R_3 & \text{otherwise} \end{cases}$$

where

```
R1 = ALL x̄ ALL ȳ ALL z̄
  CONSTRUCT GRAPH <I> { Proj_R(x̄) } WHERE { ψ(x̄,ȳ,z̄) }

R2 = ALL x̄ EX ȳ
  CONSTRUCT GRAPH <I> { Gen_R(x̄,ȳ) } WHERE GRAPH <I> { Proj_R(x̄) }

R3 = F_∀∃(ALL x̄ ALL ȳ Qz̄
  CONSTRUCT { ξ(x̄,ȳ,z̄) } WHERE { ψ(x̄,ȳ,z̄) . GRAPH <I> { Gen_R(x̄,ȳ) } } )
```

and $\mathsf{Gen}_R, \mathsf{Proj}_R, I$ are new IRIs that do not occur in P. The definition is analog for rules with graph specification in the head.

The idea of the rewriting is to extract all existential variables $\bar{y}$ that depend only on the universal variables $\bar{x}$ from ϕ. A specific *generator rule* R_{gen} states their existential dependence on $\bar{x}$ separately. To allow R_{gen} in $\forall\exists$ form, we first project all variables in ψ on only the relevant variables $\bar{x}$ in R_{proj}, the *projection rule*. Finally, we query both the original body and the generator rule in R_{join}. Since Gen is a new IRI (and thus there can be no further rules with Gen in the head) it suffices together with $\bar{x}$ to identify the corresponding $\bar{y}$.

Even though a classical logic formula with arbitrary quantifier alternation has, in general, no logical equivalent in the prefix class $\forall\exists$, this does no longer hold if we allow the extension of the vocabulary with "helper constants" that are ignored when considering equivalence. Here this is provided by the notion of default-graph equivalence introduced above. The above rewriting extends the vocabulary of the SPARQLog program in two ways:

1. It introduces, for each rewritten rule, a new graph identifier constant (I in Definition 1.16). All intermediary tuples introduced by the rewriting of that rule are constructed to belong to I (only $\xi(\bar{x},\bar{y},\bar{z})$ remains in the original graph).
2. It introduces, for each rewritten rule, two new RDF resource IRIs $\mathsf{Gen}_R, \mathsf{Proj}_R$. It would actually suffice to use introduce two such new IRIs overall, as the rewriting of different rules can reuse these constants without clash (due to the separate graph identifiers). But for clarity we use also in this case distinct new constants. In fact, the rewriting remains applicable even if no concept such as graph identifiers exists in the rule language. But in this case, we need a form of equivalence up to certain constants or predicate symbols (see, e.g., the notion of relativised equivalence in [17]).

To illustrate the rewriting consider again rule B_3 from Section 1.4:

```
ALL ?x EX ?y ALL ?z                                                    B3
CONSTRUCT { ?x eswc:heldBy ?y .            ?z foaf:knows ?y }
WHERE     { ?x rdf:type    eswc:TalkEvent. ?z eswc:attendeeAt ?x }
```

For this rule we obtain the following rewriting $F_{\forall\exists}(B_3)$ using `http../I` as IRI for the intermediary graph, `http../Gen1` as IRI for R_1 and `http../Proj1` as IRI for R_2:

```
ALL ?x ALL ?y ALL ?z
CONSTRUCT { ?x eswc:heldBy ?y . ?z foaf:knows ?y }
WHERE GRAPH <http../I> { ?x rdf:type eswc:TalkEvent. ?z eswc:attendeeAt ?x.
                <http../Gen1> rdf:_1 ?x . <http../Gen1> rdf:_2 ?y }

ALL ?x EX ?y
CONSTRUCT GRAPH <http../I> {<http../Gen1> rdf:_1 ?x. <http../Gen1> rdf:_2 ?y}
WHERE GRAPH <http../I> { <http../Proj1> rdf:_1 ?x }

ALL ?x, ?y, ?z
CONSTRUCT GRAPH <http../I> { (<http../Proj1> rdf:_1 ?x }
WHERE     { ?x rdf:type eswc:TalkEvent. ?z eswc:attendeeAt ?x }
```

Observe that the rewriting essential splits the prefix of the original rule at any $\forall$ after an $\exists$ and distributes the prefix parts over several rules. The triples with fresh IRIs allow us to link the bindings for parts of the prefix between different rules.

The $\forall\exists$ rewriting of an SPARQLog program is, if restricted to the default graph, equivalent to the original program.

Theorem 1.3. *Let P be an SPARQLog program. Then $F_{\forall\exists}(P)$ is default-graph equivalent to P.*

In other words, SPARQLog restricted to $\forall\exists$ rules is as expressive as full SPARQLog if we consider default-graph semantics.

Proof. Let R be an SPARQLog rule as in the definition of $F_{\forall\exists}$. We show that (1) $\phi(F_{\forall\exists}(R))$ FO-entails $\phi(R)$ and (2) if $A = (D, \mathrm{Rel}, \mathrm{Fun})$ is a first-order model of R then there is an extension B of A with only triples from the auxiliary relations in the auxiliary graph I that is a model of $\phi(F_{\forall\exists}(R))$. We omit sub- and superscripts if they are clear from the context. Finally, let $R_3 = F_{\forall\exists}(R'_3)$.

We first show that $\phi(F_{\forall\exists}(R))$ FO-entails $\phi(R)$. The proof is by induction on the number of quantifier alternations in R. The base case is trivial. For the induction step let $A = (D, \mathrm{Rel}, \mathrm{Fun})$ be a FO-model of $\phi(F_{\forall\exists}(R))$. To show that $A \models \phi(R)$, let $\bar{d} \in D^*$ be a sequence of domain elements with the same length as $\bar{x}$. If for all $e \in D$ and $\bar{f} \in D^*$ with $|\bar{f}| = |\bar{z}|$ it holds that $A \not\models \psi(\bar{d}, e, \bar{f})$ then we are done. Otherwise there are $e \in D$ and $\bar{f} \in D^*$ such that $A \models \psi(\bar{d}, e, \bar{f})$. As by hypothesis $A \models \phi(R_1)$ if follows that $A \models \tilde{\mathsf{Proj}}_R(\bar{d}, I)$. Therefore as $A \models \phi(R_2)$ there is an $e' \in D$, such that $A \models \tilde{\mathsf{Gen}}_R(\bar{d}, e', I)$. Finally as $A \models \phi(R_3)$ if follows from the induction hypothesis that $A \models \phi(R'_3)$ and thus $A \models \phi(R)$.

We no show that if $A = (D, \mathrm{Rel}, \mathrm{Fun})$ is a model of $\phi(R)$ then there are triples $T_1 = \{\tilde{\mathsf{Proj}}(\bar{x}, I) \ldots \tilde{\mathsf{Proj}}(\bar{x}, y, \bar{z}, I)\}$ and $T_2 = \{\tilde{Gen}(\bar{x}, I) \ldots \tilde{Gen}(\bar{x}, y, \bar{z}, I)\}$ such that the extension $B = (D, \mathrm{Rel} \cup T_1 \cup T_2, \mathrm{Fun})$ of A is a model for $\phi(F_{\forall\exists}(R))$.

The proof is by induction on the number of quantifier alternations in R. Again the base case is trivial. For the induction step let A be a model of $\phi(R)$. We define

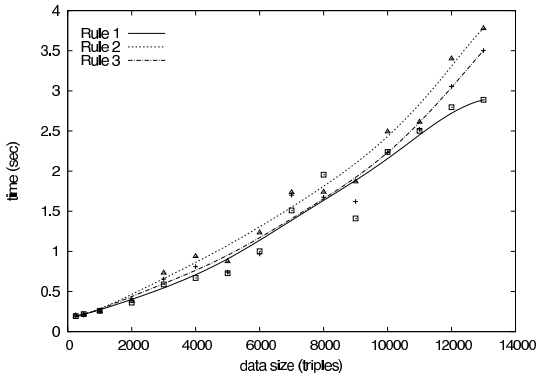

Fig. 1.3 Performance of SPARQLog on rules 1, 2 and 3

$$T_1 = \{\tilde{\mathsf{Proj}}(\bar{x},I) : \exists y,\bar{z} : \phi(\psi(\bar{x},y,\bar{z}))\}$$
$$T_2 = \{\tilde{\mathsf{Gen}}(\bar{x},y,I) : \phi(Qz : \texttt{CONSTRUCT}\ \ \xi(\bar{x},y,\bar{z})\texttt{WHERE}\ \ \psi(\bar{x},y,\bar{z}))\}$$

and $C = (D, \mathrm{Rel} \cup T_1 \cup T_2, \mathrm{Fun})$. With this definition it is a tautology that $C \models \phi(R_1)$. To show that $C \models R_2$ let $d \in D^*$. As $A \models \phi(R)$ it holds that there is an $e \in D$ such that $A \models \phi(Qz : \texttt{CONSTRUCT}\ \ \xi(\bar{x},y,\bar{z})\ \ \texttt{WHERE}\ \ \psi(\bar{x},y,\bar{z}))$. Thus $C \models \phi(R_2)$. Finally we observe that $\phi(R) \models \phi(\theta)$ where

```
θ = ALL x̄ ALL ȳ Qz̄
  CONSTRUCT { ξ(x̄,ȳ,z̄) } WHERE { ψ(x̄,ȳ,z̄) . GRAPH <I> { Gen_R(x̄,ȳ) } }
```

As C is a model of $\phi(R)$ it is also a model of $\phi(\theta)$. By the induction hypothesis there is an extension B of C that is model of $R_3 = F_{\forall\exists}(\theta)$.

1.5.3 Experimental Comparison with SPARQL Engines

The reduction of SPARQLog to standard logic programs (Section 1.4.3) allows for a direct implementation of SPARQLog on top of any logic programming or database engine that supports value invention and recursion. In the following, we we compare experimentally the performance of a very simple prototype based on that principle with two of the more common SPARQL implementations. Our implementation of SPARQLog uses a combination of Perl pre- and post-filters for Skolemisation, Unskolemisation, and normalisation of SPARQLog programs and XSB Prolog to evaluate the Skolemised programs.

We compare our implementation with the ARQ SPARQL processor of Jena (Version 2.1) and the SPARQL engine provided by the Sesame RDF Framework. For Sesame, we choose the main-memory store as it is "by far the fastest type of repository that can be used" according to Sesame's authors. With this store, Sesame becomes a main-memory, ad-hoc query engine just like SPARQLog and ARQ. As common for ad-hoc queries we measure overall execution time including both load-

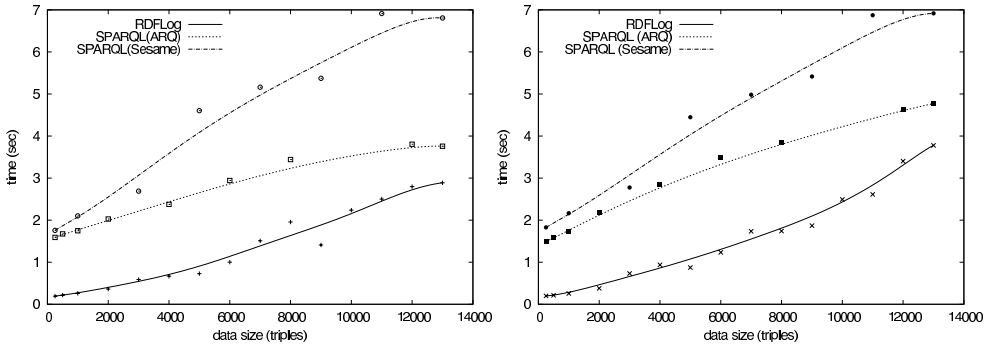

Fig. 1.4 Performance comparison on rule 1 (left) and on rule 2 (right)

ing of the RDF data and execution of the SPARQL or SPARQLog query. For the comparison, we only consider rules without existential quantification (rule 1 below) or with $\forall\exists$ quantification (rule 2 below, expressible by blank nodes in the `CONSTRUCT` graph pattern in SPARQL). Rules with arbitrary quantifier alternation are not considered as they are not expressible in SPARQL (the rewriting from Section 1.5.2 does not apply as SPARQL is single-rule and provides no projection).

In the experiments we evaluate three different queries against an RDF graph consisting of Wikipedia data. The experiments have been carried out on a Intel Pentium M Dual-Core with 1.86 GHz, 1 MB cache and 2 GB main memory. For each setting, the running time is averaged over 25 runs. We compare the following rules (with appropriate prefix definitions and dataset clauses).

```
Rule 1: ALL ?x ALL ?y
  CONSTRUCT { ?x test:connected ?y } WHERE {?x wiki:internalLink ?y }
Rule 2: ALL ?x ALL ?z EX ?z
  CONSTRUCT { ?x test:connected ?z } WHERE {?x wiki:internalLink ?y }
Rule 3: EX ?z ALL ?x ALL ?y
  CONSTRUCT { ?x test:connected ?z } WHERE {?x wiki:internalLink ?y }
```

Figure 1.3 shows the performance of SPARQLog for each of the rules. Note that the running time increases from rule 1 to rule 3 and from rule 3 to rule 2. The difference between rule 1 and rule 3 might be due to overhead of Skolemisation, Unskolemisation and normalization. The running time difference between rule 3 and rule 2 may be attributed to the lower amount of blank nodes generated in rule 3, as the existential quantifier is outside of the scope of all universal quantifiers.

Figure 1.4 compares the performance of SPARQLog with that of ARQ and Sesame for rule 1 and rule 2 (we omit rule 3 as it is not expressible in SPARQL). Despite its light-weight, ad-hoc implementation, SPARQLog outperforms ARQ and Sesame in this setting. The figures show moreover that also for ARQ and Sesame, blank node construction does not bear any significant additional computational effort.

1.6 Conclusion

Blank nodes are one of RDF's distinguishing features. Yet they have been entirely neglected or treated only in a limit fashion in previous approaches to RDF querying. With SPARQLog we advance the knowledge about the combination of blank nodes and rules (and thus RDF and rules) in three directions: (1) We show that restrictions of RDF wrt. blank nodes occurrence can be treated in a semantics based purely on entailment. (2) Though unrestricted combinations of recursive rules and blank nodes in rule heads lead, unsurprisingly, to a undecidable, Turing-complete language, we identify a large fragment of such rules that is still decidable. This fragment is strictly larger than previous decidable languages with recursive rules and blank nodes in the head. (3) Finally, we show that quantifier alternation does not add to the expressiveness or complexity of a language with $\forall\exists$ rules and projection. The latter form of rules is commonly found in data exchange or SPARQL rule extensions. In other words, quantifier alternation comes for free for such languages.

Though we present the results here in the context of RDF querying, they apply to a wide range of logic languages with horn rules extended by existential quantification. In particular, in data exchange such languages are common but mostly limited to $\forall\exists$ rules.

References

1. Broekstra, J., Kampman, A.: An rdf query and transformation language. In: Semantic Web and Peer-to-Peer. Springer (2006) 23–39
2. Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: Rdflog: It's like datalog for rdf. In: Proc. Workshop on (Constraint) Logic Programming (WLP). (2008)
3. Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: Taming existence in rdf querying. In: Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR). (2008)
4. Deutsch, A., Nash, A., Remmel, J.: The chase revisited. In: Proc. ACM Symp. on Principles of Database Systems (PODS), New York, NY, USA, ACM (2008) 149–158
5. Ebbinghaus, H.D., Flum, J., Thomas, W.: Mathematical Logic. Springer (1994)
6. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: Semantics and query answering. In: Proc. Int'l. Conf. on Database Theory (ICDT). (2003) 207–224
7. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: Getting to the core. ACM Transactions on Database Systems **30**(1) (2005) 174–210
8. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. **30**(4) (2005) 994–1055
9. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF Querying: Language Constructs and Evaluation Methods Compared. In: Tutorial Lectures Int'l. Summer School 'Reasoning Web'. Volume 4126 of Lecture Notes in Computer Science., Springer (2006) 1–52 19 citations [Google Scholar].
10. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of semantic web databases. In: Proc. ACM Symp. on Principles of Database Systems (PODS), New York, NY, USA, ACM Press (2004) 95–106
11. Hayes, P., McBride, B.: Rdf semantics. Recommendation, W3C (2004)
12. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: Rql: a declarative query language for rdf. In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM Press (2002) 592–603

13. Marnette, B.: Generalized schema-mappings: From termination to tractability. In: Proc. ACM Symp. on Management of Data (SIGMOD), Providence, RI, USA, ACM Press (2009)
14. Meier, M., Schmidt, M., Lausen, G.: Stop the chase. CoRR **abs/0901.3984** (2009)
15. Muñoz, S., Pérez, J., Gutierrez, C.: Minimal deductive systems for rdf. In: Proc. European Semantic Web Conf. (ESWC). Volume 4519 of Lecture Notes in Computer Science., Springer (2007) 53–67
16. Noy, N., Rector, A., Hayes, P., Welty, C.: Defining n-ary relations on the semantic web. Working group note, W3C (2006)
17. Oetsch, J., Tompits, H., Woltran, S.: Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. In: Proc. Nat'l. Conf. on Artificial Intelligence (AAAI). (2007) 458–464
18. Polleres, A.: From sparql to rules (and back). In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM (2007) 787–796
19. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. Proposed recommendation, W3C (2007)
20. Rosati, R.: The limits and possibilities of combining description logics and datalog. In: Proc. Int. Conf. on Rule Markup Languages (RuleML). (2006) 3–4
21. Schenk, S., Staab, S.: Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM (2008) 585–594
22. Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. CoRR **abs/0812.3788** (2008)
23. Sintek, M., Decker, S.: Triple—a query, inference, and transformation language for the semantic web. In: Proc. Int'l. Semantic Web Conf. (ISWC). (2002)
24. ter Horst, H.J.: Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web **3** (2005)
25. Yang, G., Kifer, M.: Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. Journal of Data Semantics **1** (2003) 69–97