Semantic Web Journal
XX(X):1–23

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

SPARQL in N3: Translating SPARQL into a Semantic Rule Language to Facilitate Logic Reasoning on the Semantic Web

Dörthe Arndt[1,2], William Van Woensel[3], and Dominik Tomaszuk[4]

Abstract

Rule-based reasoning on the Semantic Web (SW) is a logics-based paradigm for authoring expressive, exchangeable and interoperable rules. Such rules have the ability to natively access, operate on, and reason over online and interconnected data sources. While the full potential of rule-based reasoning has not yet been tapped, query-based data access is well developed and mature on the SW. The SPARQL Protocol and RDF Query Language (SPARQL) is a well-known and widely used standard for querying SW data. An option is thus to express rule-based reasoning using SPARQL queries to leverage its popularity and ease-of-use. Prior approaches to utilize SPARQL queries in this way are either limited in expressivity, operational semantics (bottom-up reasoning), or rely on a target rule language with a significant impedance mismatch (Datalog). We present an approach that instead translates SPARQL queries to the Notation3 (N3) SW rule language. We adhere to a one query–one rule principle, where each SPARQL query is translated into a single N3 rule, in order to improve the exchangeability of translated rules. We formally prove the correctness of our approach and evaluate the performance of our approach compared to the state of the art. We view our work as a first step in the cross-pollination of SW query and rule languages.

Keywords

SPARQL, Notation3, rule-based reasoning

1 Introduction

The full potential of logics-based reasoning on the Semantic Web (SW) is yet to be realized. Most SW reasoning takes place using OWL2 DL (*1*) ontologies, and thus relies on Description Logics (DL). DL is however less expressive than first-order logic, and it can be challenging to express rule-based logic therein. The Semantic Web Rule Language (SWRL) (*2*) extends OWL interpretations with variables and Horn-like rules. However, it operates on OWL rather than the Resource Description Framework (RDF) (*3*), the cornerstone of the SW, and it cannot generate blank nodes. The Rule Interchange Format (RIF) (*4*) offers 3 dialects to cover the needs of rule systems, but similarly exists outside of the RDF data model. Notation3 (N3) (*5*) is a SW rule language that follows the RDF data model, extending it with variables and graph terms. To the best of our knowledge, however, N3 (and RIF) are not widely adopted.

In contrast, querying on the SW has been well established. The SPARQL Protocol and RDF Query Language (SPARQL) (*6*) is a widely used standard for querying RDF graphs, offering useful features such as filter functions, aggregation, grouping, and property paths. Due to its syntactic similarity to SQL, it provides an easy-to-learn syntax for those who have worked with relational databases.

We point out that, when writing a SPARQL CONSTRUCT query (e.g., Listing 4), authors are in fact creating a logic rule. If their WHERE clause holds, then the query template is instantiated and returned as RDF triples. In logic rule terms, an inference is made. When CONSTRUCT queries refer to the instances produced by themselves or other queries, one can see they could serve as rules. The use of SPARQL to express logic rules has the following potential benefits. Firstly, it may help *realize the potential of logic SW reasoning* by leveraging the popularity and expressivity of SPARQL. Authors are not asked to learn a new rule language, but rather re-use a well-known and well-documented query language. Secondly, it *expands SPARQL itself with general (or full) recursion* (*7*), since it would allow a CONSTRUCT query to act on its own results or the results of other queries.

[1]Computational Logic Group, Technische Universität Dresden, Dresden, Germany
[2]ScaDS.AI, Dresden/Leipzig, Germany
[3]Telfer School of Management, University of Ottawa, Ottawa, ON, Canada
[4]University of Bialystok, Bialystok, Poland

Corresponding author:
Dörthe Arndt, doerthe.arndt@tu-dresden.de

While SPARQL already supports a type of *limited (or partial) recursion* in the form of property paths (steps can act on the result of prior steps), it is limited in expressivity (*8*). Recursion also improves modularity, since reusable logic parts can be encapsulated in separate queries (*9*). Finally, it can provide ideas for the *cross-pollination of SW query and rule languages*, which tend to rely on wholly different semantic definitions—causing significant complexity when implementing some constructs—while query vs. rule engines offer performance benefits for different use cases.

We are not the first to propose the use of SPARQL as a logic rule language. We have observed two categories of approaches. Approaches in the first category have translated SPARQL into a non-SW logic rule language, namely Datalog. In doing so, the rule language's reasoning machinery can be used for reasoning over the queries. However, similar to SWRL and RIF, there is again an impedance mismatch; Datalog expresses everything using predicates, whereas the RDF data model uses triples instead. Datalog also does not follow SW principles on exchangeability and interoperability, as rules are designed for local use (e.g., Datalog predicate names are not meant to be unique across systems). In the second category, a SPARQL query engine is extended with reasoning capabilities. While query engines are excellent at dealing with large numbers of joins and large datasets, they lack top-down (backward) reasoning capabilities.

Similar to the first category, we propose a translation of SPARQL, but instead target the N3 SW rule language. In contrast to Datalog, N3 follows the RDF datamodel, and adheres to SW principles on exchangeability and interoperability. To improve upon these principles, we further engineered our translation around a *one query–one rule* principle: each SPARQL query is translated to a single N3 rule, which improves the traceability and exchangeability of translated rules. This is made possible by a "runtime" ruleset, which implements SPARQL features that would otherwise require multiple rules. Compared to query engine extensions, existing N3 reasoning machinery can be leveraged to implement both bottom-up (forward) and top-down (backward) reasoning. Notably, as an illustration of the meta-reasoning[1] capabilities of N3, we use N3 itself to implement the translation from a triple-based SPARQL (SPIN (*10*)) to N3. We refer to our rule translation approach as SiN3 (SPARQL in N3). While the focus lies on CONSTRUCT queries, SiN3 also supports SELECT queries. We formally prove the correctness of the translation, and present the results of a performance evaluation comparing SiN3 with other recursive SPARQL approaches. All code and artifacts, together with a demo, is available online (*11*)

As mentioned, we foresee our work to represent a first step in the cross-pollination between SW query and rule languages. When consequential, we highlight the semantic mismatch between a query language like SPARQL, where semantics are defined by *variable mappings*; and a logic rule language like N3, which relies on model-theoretic semantics (interpreting graphs through models), meaning that systems instead deal with *instantiated patterns*. Moreover, as a first, albeit small, step in their cross-fertilization, the "runtime" ruleset allows certain SPARQL features (union, property paths) to be directly used within N3. Our empirical evaluation further contrasts the performance benefits of query vs. rule engines for different setups: query engines have a clear benefit when dealing with many joins and extensive datasets, but top-down (rule-based) reasoners are better equipped at dealing with huge search spaces. The evaluation further presents a health informatics use case that demonstrates the added value of expanding SPARQL with recursion via rule-based reasoning: *modularity*, by encapsulating reusable logic in separate queries; and *expressivity*, by using a custom entailment regime (OWL2 RL (*12*)).

In summary, we make the following contributions:

- **(C1)** The SPARQL-in-N3 (SiN3) translation that applies a *one query–one rule* principle, i.e., each SPARQL query is compiled into a single N3 rule. To that end, we present the novel concept of a "runtime" ruleset.
- **(C2)** SiN3 adds full recursivity to SPARQL; we illustrate the associated benefits of modularity and expressivity via a healthcare use case.
- **(C3)** We provide a formal proof of the correctness of the translation.
- **(C4)** We report an empirical evaluation that compares with state-of-the-art approaches, and release public code and artifacts, including a runnable demo.
- **(C5)** We present a first step in the cross-pollination of SW query and rule languages, which includes highlighting their semantic mismatches, contrasting performance benefits, a "runtime" ruleset and SPARQL recursion.

Prior work includes a demo paper (*9*) and a conference paper (*13*) that introduced the initial SiN3 approach. This paper builds upon (*13*) with an improved and much more comprehensively described translation, which now includes support for property paths and improved handling of complex constructs (OPTIONAL and MINUS); expanded logical proofs of the correctness of the translation; and an expansion of the related work and evaluation sections. The remainder of this paper is as follows: Section 2 presents background on RDF, SPARQL, and N3. Section 3 discusses related work on the use of SPARQL as a logic rule language. Section 4 provides a formal definition of SPARQL, and Section 5 provides the same for N3. Section 6 presents the SiN3 approach to translating SPARQL into N3. Section 7 presents performance evaluation results. Finally, Section 8 presents conclusions and future work.

[1] By analogy with meta-programming in procedural languages.

```
<http://example.org/John>
    <http://example.org/hasAge> "30" .
```

Listing 1: An RDF triple representing an individual's age

```
@prefix ex: <http://example.org/> .
ex:John ex:name "John Doe" .
ex:John ex:hasAge 30 .
ex:John ex:hasEmail
    <mailto:john.doe@example.org> .
```

Listing 2: A graph describing an individual with name, age, and email.

2 Background

This section provides a high-level overview of RDF, SPARQL, and N3. Subsequent sections detail the semantics of the latter two (Sections 4 and 5).

RDF . A resource is anything (e.g., abstract concept or physical object) that can be described. Resources are identified using IRIs (Internationalized Resource Identifier) and literal terms (e.g., numbers and strings). They are described using triples (or statements), where the *subject* represents the resource being described; the *predicate* (or property) the attribute or relationship of the subject; and the *object* the attribute's value or a related resource. For example, the RDF triple in Listing 1 describes the resource John. The subject is `<http://example.org/John>`, an IRI identifying John; the predicate is `<http://example.org/hasAge>`, an IRI identifying the "has age" attribute; and the object is `30`, a literal that represents the person's age. For brevity, an IRI can be written as a qualified name, with a prefix and local name separated by ":", such as `ex:John`. The prefix refers to a namespace, which is defined as shown in Listing 2 (line 1).

Collectively, RDF triples represent a *graph*, with resources as nodes and predicates as edges[2]. Consider the graph in Listing 2, which further describes John by adding his full name and email address. Here, John represents a node with 3 outgoing edges, namely `ex:name`, `ex:hasAge`, and `ex:hasEmail`; each of which point to a leaf node representing a literal term ("John Doe", 30) or IRI (`mailto:`[...]).

SPARQL . A SPARQL SELECT query formulates patterns to be matched against RDF triples. To retrieve the age of John, one might write a SELECT query as shown in Listing 3. In the WHERE clause, a triple pattern will match any triple with subject `ex:John` and predicate `ex:hasAge`, and map the triple's object to the `?age` variable. This gives rise to a solution mapping for `?age`, which is discussed in more detail in Section 4.2. A SELECT query then returns the values mapped to the listed variable `?age`.

```
PREFIX ex: <http://example.org/>
SELECT ?age
WHERE { ex:John ex:hasAge ?age }
```

Listing 3: A SPARQL SELECT query to retrieve John's age.

```
PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.org/>
CONSTRUCT { ?person rdf:type ex:Adult }
WHERE { ex:John ex:hasAge ?age .
        FILTER (?age > 18) }
```

Listing 4: A SPARQL CONSTRUCT query that creates an RDF graph.

```
@prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix math:
  <http://www.w3.org/2000/10/swap/math#>.
@prefix ex: <http://example.org/> .

ex:John ex:hasAge "30" .
ex:Jane ex:hasAge "25" .
ex:Bob ex:hasAge "17" .

{ ?person ex:hasAge ?age .
  ?age math:greaterThan 18 .
} => { ?person rdf:type ex:Adult } .
```

Listing 5: An N3 rule for inferring adulthood based on age

A CONSTRUCT query instead instantiates and returns an RDF graph. For instance, the CONSTRUCT query in Listing 4 instantiates the template graph saying that John is (`rdf:type`) an adult in case the WHERE clause holds. The latter clause includes the same triple pattern as before, together with a FILTER expression that checks whether the found age ¿ 18.

N3. N3 is a superset of RDF, as it includes all elements from RDF and adds extra ones, such as universal variables and graph terms (*5*). It also allows declaring logical implications, or rules. Consider the N3 rule in Listing 5, i.e., a triple with, as subject, a graph term that represents the rule body; as predicate, the => symbol[3]; and as object, a graph term representing the rule head. The rule head is inferred when the rule body holds. The example rule body includes a triple pattern that matches any triple describing a person's age. In this case, the triple's object is used to instantiate the `?age` variable. A second triple pattern checks, using

[2]Technically, a predicate (e.g., `hasAge`) may occur as a subject or object as well, so it could also occur as a node.

[3]This symbol is a shortcut for the `log:implies` predicate.

```
ex:John rdf:type ex:Adult .
ex:Jane rdf:type ex:Adult .
```

Listing 6: Inferred adulthood using an N3 rule engine

the `math:greaterThan` builtin, whether the instantiated `?age` variable is greater than 18. If so, the rule head infers that the person is (`rdf:type`) an adult. Using this rule and the data, an N3 rule engine, such as eye (*14*), can infer the information shown in Listing 6.

Listing 4 and 5 illustrate the close relation between the syntax of SPARQL and N3, which will facilitate the translation between them.

3 Related Work

The idea of using SPARQL for rule-based reasoning is not new. We have observed two approaches: those that translate SPARQL into logic rule languages such as Datalog (Section 3.1); and those that extend query engines with reasoning capabilities (Section 3.2). A summary of the proposals described below is provided in Table 1.

3.1 Translating SPARQL to Logic Rule Languages

To the best of our knowledge, Polleres (*15*) is the first to translate SPARQL into Datalog with negation as failure based on Answer Set Programming (ASP). The author proposes the use of blank node-free CONSTRUCT queries as a logic rule language for RDF. By translating such queries into Datalog, SPARQL can be implemented using existing Datalog rule engines. Polleres points out that subtleties, including OPTIONAL clauses, are not straightforward to translate to Datalog. The author presents 3 possible semantics for SPARQL, each defining the compatibility of variable mappings (and therefore joins) in an increasingly strict way. The paper describes the translation to Datalog for each of these 3 semantics. The author implemented their translation using dlvhex, a framework for developing DLV extensions [4]. The README notes that the system does not yet support CONSTRUCT queries and solution modifiers, and that FILTERs are still incomplete.

Gottlob et al. (*8*) note that there are still useful queries that cannot be expressed in SPARQL 1.1 due to the lack of full recursion. As requirements for a query language, they mention reasoning, navigational and full recursive capabilities, as well as tractable query evaluation. Due to the full recursion, such a query language can be used to represent logic rules, and associated query engines to implement rule-based reasoning. While TriQ-Lite (*25*) based on $Datalog^{\exists,\neg s,\bot}$ meets these desiderata, it does not offer a simple and non-compositional syntax. Hence, the authors introduce TriQ-Lite 1.0, with a simpler syntax and semantics with tractable efficiency, based on Warded $Datalog^{\exists,\neg sg,\bot}$. SPARQL queries, following the OWL2 QL (direct semantics) entailment regime, are translated into TriQ-Lite 1.0. As far as we know, the approach is not implemented.

Angles et al. (*7*) introduce SparqLog, a system that translates SPARQL into Warded $Datalog\pm$ (*25*) and subsequently executes the Datalog rules using Vadalog (*26*). This Datalog variant is described as the best compromise between complexity and expressiveness, and also supports existential quantification. Vadalog further supports ontology reasoning, namely the OWL2 QL entailment regime, out of the box. It is thus worth noting that both Gottlob et al. and Angles et al. rely on Warded Datalog and support OWL2 QL, which depends on existential quantification. SparqLog lacks support for the CONSTRUCT, BIND, and VALUES constructs. Datalog's support for recursion is used to implement the partial recursion in SPARQL 1.1 (i.e., property paths). However, in contrast to Polleres and Gottlob et al., their goal is not to allow the use of SPARQL as a logic rule language, and they do not add full recursion to SPARQL. The authors experimentally validated their system to check compliance with SPARQL 1.1, and measure its performance using SPARQL engine benchmarks.

These translations operate differently from the proposed approach (SiN3). As mentioned, Datalog expresses everything using predicates, whereas RDF expresses everything as a triple. This has multiple repercussions: (a) Datalog translations rely on predicates with arbitrary arities, which are not directly supported in N3 as it is centered on triples; (b) a separate data translation step is required to generate Datalog facts from RDF triples (*25*). Furthermore, these translations do not adhere to a one query–one rule principle, as they may expand a single query into multiple rules: for example, Angles et al. (*25*) translate a single query, with a 1-step property path and FILTER expression, into 4 Datalog rules. These differences motivated the need for the separate translation approach presented in this paper.

Finally, as mentioned above, the authors of these papers either do not focus on using CONSTRUCT queries as a logic rule language (*7*), or did not (fully) implement their approach (*8*, *15*). As a result, it is also not suitable, or possible, to compare their performance with our approach.

3.2 Extensions to SPARQL Engines

This section discusses works that add reasoning capabilities to SPARQL engines, albeit with full recursion (Section 3.2) or limited recursion for graph navigation (Section 3.2). We found the latter to be mostly targeted by earlier work (2007, 2010), i.e., before the introduction of property paths in SPARQL 1.1 (2013) (*6*). Since we also target full recursion

[4] http://www.dlvsystem.com/

Work / System	Cat.	Rec.	Syn.	Qry.	RDF-n.	Impl.	Eval.
Polleres (15)	S2R	F	N	S,C,A	N	–	–
TriQ-Lite 1.0 (8)	S2R	F	N	–	N	–	–
SparqLog (7)	S2R	PP	N	S,A	N	Y	Y
recSPARQL (16, 17)	Eng.	F	Y	C (+S/A)	Y	Y	Y
SPIN / SpinRDF (10, 18)	Eng.	–	N	C	Y	Y	–
SPARQAL (19)	Eng.	F	Y	S,A	Y	Y	Y
LDScript (Corese) (20)	Eng.	F	Y	S,C (+ext.)	Y	Y	Y
Atzori et al. (21)	Eng.	–	Y	–	–	–	–
nSPARQL (22)	Nav.	Nav	Y	–	Y	–	–
SPARQLeR (23)	Nav.	Nav	Y	S,C,A	Y	Y	Y
TriAL* (24)	Alg.	Nav	N	–	–	–	–
SiN3	Eng.	F	N	S,C	Y	Y	Y

Table 1. Paper-style comparison of approaches. *Legend:* Cat.: S2R = SPARQL-to-rules translation; Eng. = engine/query-layer extensions; Nav. = navigational recursion; Alg. = algebraic/theoretical formalism. Rec.: F = full recursion; PP = property paths / limited recursion; Nav = navigational recursion; – = not the focus/unspecified. Syn.: extra SPARQL syntax required (Y/N). Qry.: S = SELECT, C = CONSTRUCT, A = ASK (mainly). RDF-n.: operates natively on RDF without an RDF→facts translation step (Y/N). Impl./Eval.: availability and whether an experimental evaluation is reported (Y/–).

for SPARQL, we discuss the former category in more detail and discuss, among other things, the following:

Extra SPARQL syntax. Others introduce extra syntax for full recursion (16, 17, 19, 21) or imperative code constructs (20). We note that SiN3 does not require additional syntax; any SPARQL triple pattern can refer to other query results.

Types of queries and their restrictions. Some works focus solely on `CONSTRUCT` queries (16, 17), while others target `SELECT` queries (19). Some impose restrictions on `CONSTRUCT` queries to guarantee a fixed point (16, 17). SiN3 supports both CONSTRUCT and SELECT queries.

Implementation and performance evaluations. We list the location of available implementations, and indicate whether performance evaluations were performed. As mentioned, all relevant SiN3 code and artefact can be found online (11), and our evaluation is found in Section 7.

Full recursion. Kostylev et al. (16, 17) present recSPARQL, which introduces extra syntax for a fixed point operator. This operator, which resembles a similar operator from the SQL:1999 standard, iteratively populates a temporal graph (treated as a named graph) via a CONSTRUCT query, until a least fixed point is obtained. Other CONSTRUCT queries, as well as the above CONSTRUCT query, can themselves use GRAPH patterns[5] to refer to the temporal graph to act on prior query results. A SELECT or ASK query can similarly refer to the temporal graph to return a final result. RecSPARQL allows specifying a max. recursion depth, noting that very long paths are seldom of interest (17). The focus lies on CONSTRUCT queries lacking blank nodes in the query template, and limitations are imposed on CONSTRUCT queries to always have fixed points. This requires monotone queries, i.e., without explicit negation and no recursive references in OPTIONAL parts (called semi-positive). The proposal is equal in expressive power to $Datalog^{rbr}_{rdf}$ (rbr = rule-by-rule stratification). In follow-up work (17), authors add a restriction to linear recursion, i.e., where iteration *i* only relies on the original dataset and new triples from iteration *i-1*. The authors mention that this resembles design choices from most commercial SQL systems and graph databases. The authors implemented recSPARQL on top of Apache Jena (27) (ARQ module). The implementation is available online[6]. The system was evaluated on 3 datasets.

The SPARQL Inferencing Notation (SPIN) is a W3C Member Submission (18). It presents an RDF-based SPARQL syntax, i.e., which encodes SPARQL queries using RDF triples. In fact, our SiN3 translation starts from this RDF representation, as it allows N3 rules to read input SPARQL queries as triples (9). SPIN further offers a modeling vocabulary to, among other things, specify inference rules using CONSTRUCT queries. The SpinRDF reasoner, built on top of Apache Jena and available on GitHub (10), can reason over these inference rules to infer new triples.

Hogan et al. introduce SPARQAL (19) that integrates graph querying with analytical tasks. The authors introduce extra syntax for DO…WHILE loops: such a loop executes statements until a termination condition is met; either the least fixed point being reached, max. recursion depth, or a custom condition (ASK query). Other syntax allows assigning SELECT query results to solution variables; queries can refer to solution variables to act on prior query results. The authors prove Turing completeness. To allow acting on prior query results that may not fit in memory, joins can be performed using a batch-based technique similar to Map-Reduce. SPARQAL was implemented on top of Apache Jena (27), and the system is available online[7]. SPARQAL was evaluated on Wikidata.

Corby et al. introduce LDScript (20), which adds a set of imperative programming constructs on top of SPARQL, including functions and loops. Functions can hereby contain

[5] https://www.w3.org/TR/sparql11-query/#queryDataset
[6] https://adriansoto.cl/RecSPARQL/
[7] https://github.com/VHDG88FKL/SPARQL-Analytics

any SPARQL expression, including IF and EXISTS, and functions can be referenced in a query's FILTER. The online documentation[8] illustrates how LDScript functions could be used to implement full recursion. LDScript was implemented on top of the SPARQL Corese engine (*28*). The system performance was compared with equivalent native Java/JS code. The implementation is part of Corese.

Atzori et al. (*21*) introduce a `wfn:runSPARQL` function for resolving recursive expressions. A recursive expression is written as a string, which includes recursive calls to `wfn:runSPARQL` passing the same string and new variable values. The expression can act on the recursive call's result. Examples include an IF construct, which, if a termination condition is not met, recursively calls `wfn:runSPARQL`; otherwise, a value is returned. An outer BIND assigns the result of the IF construct to a "result" variable; this is what is returned by the recursive call. Based on the examples, the approach does not seem general enough to implement full recursion.

We compare the performance of SiN3 with recSPARQL (*17*) and SpinRDF (*10*) (Section 7), as these approaches are the most similar to us; they rely on CONSTRUCT queries and do not introduce imperative programming constructs.

Partial recursion. Perez et al. (*22*) present nSPARQL (nested SPARQL) (*22*) for navigating RDF graphs. To that end, triple patterns can have (nested) regular expressions in predicate position. Such a regular expression allows navigating a graph via their edges and nodes, possibly placing restrictions on their labels. The syntax is similar to XPath (*29*); for instance, `next::<edge>` gets the next nodes via `<edge>`, and `edge::<node>` gets outgoing edges to `<node>`. One can construct paths: e.g., `next::a / next::b` returns nodes reachable by following edge `a` and then `b`. Expressions can also be nested using "`[]`": e.g., `next::[next::a]` involves resolving the nested expression and using it in the outer expression. The authors show that their language is sufficiently expressive to implement RDFS semantics for SPARQL queries.

Kochut et al. introduce SPARQLeR (SPARQL Extended with Regular paths) (*23*), where triple patterns can have `%path` variables as predicates; a path variable matches any path between two nodes. For instance, the triple pattern `<r> %path ?res .` identifies all nodes reachable from subject `<r>`. The individual elements of a path (nodes and edges) can also be returned. Restrictions can be put on paths, e.g., on their edges and length. The syntax is meant to introduce only minimal changes to SPARQL syntax and semantics.

Libkin et al. (*24*) note that navigational languages, such as nSPARQL (*22*), are not fully suitable for RDF triples: they do not support predicates acting as subjects/objects of other triples, which is allowed in RDF. They present TriAL* (Triple ALgebra with recursion), which allows joins between sets of triples, with any join conditions on their subject/predicate/object (s/p/o). For example, consider the join expression $E \bowtie^{1,3',3}_{2=1'} E$, with numeric triple indexes matching a first (s_1, p_2, o_3) and a second $(s_{1'}, p_{2'}, o_{3'})$ set of triples E. The join expression subscript indicates that the first triple's predicate (p_2) is joined with the second triple's subject $(s_{1'})$. Join results can further be arbitrarily composed from the original triples; the superscript indicates that results will be composed of $s_1, o_{3'}, o_3$. To identify paths of any length, TriAL* supports recursive self-joins (left and right Kleene closures) until a fixed point is reached. For instance, in a recursive left self-join, iteration *i* involves a join between the original triple set and result of the join of iteration *i-1*. The join result is the union of all iterations. The authors present a fragment of Datalog that captures TriAL*. As far as we know, there is no concrete syntax for TriAL*.

4 SPARQL

Our definitions in this section follow the work of Pérez et al. (*30*), and extend them to accommodate the features introduced by SPARQL 1.1, in line with other work (*31, 32*).

4.1 Syntax

In SPARQL, atomic terms come from the pairwise disjoint infinite sets I, B, L, and V, representing IRIs, Blank Nodes, RDF literals and Variables, respectively. A filter condition R is a logical formula further described below. The set GP of Graph Patterns is defined recursively as follows:

(1) If P is a finite set such that $P \subseteq (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, then $P \in GP$. Each element $t \in P$ is called a *triple pattern*[9]. A set P as above is called a *Basic Graph Pattern (BGP)*.

(2) If $P, P_1, P_2 \in GP$ and R is a filter condition, then:
- AND(P_1, P_2), UNION(P_1, P_2), and MINUS(P_1, P_2) belong to GP.
- FE(P_1, P_2) and FNE(P_1, P_2) belong to GP (FE stands for FILTER EXISTS, FNE for FILTER NOT EXISTS),
- OPT$(P_1, P_2, R) \in GP$ (OPT is short for OPTIONAL)[10],
- FILTER$(P, R) \in GP$.

(3) If $P \subset (I \cup L \cup V) \times PP \times (I \cup L \cup V)$ and PP is a property path, then $P \in GP$.

Filter conditions. A SPARQL filter condition R is formed by combining elements from the set of atomic terms $I \cup B \cup L \cup V$, including $\top$ (true) and $\bot$ (false), logical connectives, equality and other comparison symbols, unary predicates

[8] http://ns.inria.fr/sparql-extension/#usecase4

[9] In practice, triple patterns allow blank nodes as subjects and objects. These are omitted here to keep the semantics concise. Our translation in Section 6 also works with blank nodes in BGPs, as they are locally scoped in N3.

[10] If there is no FILTER condition, $\top$ is added as a filter argument.

such as `bound`, `isBlank`, and `isIRI`, along with other features. It evaluates to true, false or error depending on its elements. We refer to Pérez et al. (30) for details.

Property paths. A property path PP specifies an abstract route of arbitrary length between two nodes, and can match multiple concrete paths. Property paths include the following, where p is a single predicate IRI:

- A single property p,
- An inverted property path $\hat{}PP$,
- Sequential property paths $PP_1 \,/\, PP_2$,
- Alternative property paths $PP_1 \mid PP_2$,
- Transitive closures of a property path PP^*, PP^+,
- An optional property path $PP^?$,
- A negated property $!p$, which can also be inverse $!\hat{}p$,
- A negated set of (inverted) properties $!(p_1|\ldots|p_i|\hat{}p_j|\ldots|\hat{}p_n)$.

FE and FNE. We define FE (Filter Exists) and FNE (Filter Not Exists) as separate graph patterns, even though the SPARQL specification presents them as types of filter conditions. The reason for this is that their evaluation depends not only on the variable bindings, but also the graph the query is evaluated against (Section 4.2). All other filter conditions only depend on the former (6, Section 17.4.1.4).

Sub patterns and super patterns. For each graph pattern $P \in GP$ we define the set $sub(P)$ of its sub patterns:

- if P is a BGP, then $sub(P) = \emptyset$,
- if P is of the form $X(P_1, P_2)$ with $X \in \{\text{AND}, \text{UNION}, \text{MINUS}, \text{FE}, \text{FNE}\}$, then $sub(P) = \{P_1, P_2\} \cup sub(P_1) \cup sub(P_2)$,
- if P is of the form $\text{OPT}(P_1, P_2, R)$ then $sub(P) = \{P_1, P_2\} \cup sub(P_1) \cup sub(P_2)$,
- if P is of the form $\text{FILTER}(P_1, R)$, then $sub(P) = \{P_1\} \cup sub(P_1)$.

Note that for all $P \in GP$, we have that $sub(P) \subset GP$.

Similarly to the sub patterns we can define the set of super patterns of a pattern within another pattern. Given a pattern $P \in GP$ and a sub pattern $Q \in sub(P)$. We define the super pattern $sup(Q, P)$ of Q in P as follows:

$$sup(Q, P) := \{R \in sub(P) | Q \in sub(R)\}$$

For each pair of graph patterns $P, Q \in GP$, we define the set of its *pattern difference* $pd(P, Q)$ as follows:

- if $P = Q$ or P is a BGP, $pd(P, Q) = \emptyset$
- else,
  - if P is of the form $X(P_1, P_2)$ with $X \in \{\text{AND}, \text{UNION}, \text{MINUS}, \text{FE}, \text{FNE}\}$, then $pd(P, Q) = \{P_1, P_2\} \cup pd(P_1, Q) \cup pd(P_2, Q)$,
  - if P is of the form $\text{OPT}(P_1, P_2, R)$ then $pd(P, Q) = \{P_1, P_2\} \cup pd(P_1, Q) \cup pd(P_2, Q)$,
  - if P is of the form $\text{FILTER}(P_1, R)$, then $pd(P, Q) = \{P_1\} \cup pd(P_1, Q)$.

Note that for all $P, Q \in GP$ we have $pd(P, Q) \subseteq sub(P)$.

4.2 Semantics

Let $T = I \cup B \cup L$ be the set of non-variable terms. The semantics of SPARQL relies on the mapping of variables V to non-variable terms T. A mapping μ from V to T is a partial function denoted as $\mu : V \to T$. The domain of μ, denoted as $\text{dom}(\mu)$, is the subset of V where μ is defined. Two mappings, μ_1 and μ_2, are compatible, written as $\mu_1 \sim \mu_2$, if, for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, $\mu_1(x) = \mu_2(x)$. I.e., μ_1 and μ_2 map the same variables to the same terms. We consider mappings as sets of pairs (v, t) with $v \in V$ and $t \in T$, and use the union operator $\cup$ to combine them.

Operations on sets of mappings. Given sets of mappings Ω_1 and Ω_2, we define:

(1) $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$,
(2) $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$,
(3) $\Omega_1 \setminus_M \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2 : \mu \not\sim \mu' \text{ or } \text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset\}$.

I.e., the join (1) combines all mappings that are compatible, the union (2) adds all mappings to a single set, and the difference (3) subtracts all compatible mappings, leaving only non-compatible mappings.

Applications of mappings. Let Q be a BGP or filter condition, and $var : GP \to 2^V$ be a function which maps a graph pattern to the set of all its variables. We apply a mapping μ to Q, denoted by $Q\mu$, by replacing all $v \in dom(\mu) \cap var(Q)$ with $\mu(v)$. I.e., variables in Q are instantiated with concrete terms from μ. If $R\mu = \top$ for filter condition R and mapping μ, we also write $\mu \vDash R$.

Evaluation of SPARQL graph patterns The evaluation of a graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G$, produces a set of mappings μ. It is defined as follows:

(1) If P is a BGP, then $[\![P]\!]_G = \{\mu \mid dom(\mu) = var(P) \text{ and } P\mu \subseteq G\}$,
(2) If P is $\text{AND}(P_1, P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
(3) If P is $\text{OPT}(P_1, P_2, R)$, then

$$\begin{aligned}[\![P]\!]_G = \{&\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G,\ \mu_2 \in [\![P_2]\!]_G,\\ &\mu_1 \sim \mu_2 \text{ and } \mu_1 \cup \mu_2 \vDash R\}\\ \cup \{&\mu_1 \in [\![P_1]\!]_G \mid \nexists \mu_2 \in [\![P_2]\!]_G,\\ &\mu_1 \sim \mu_2 \text{ and } \mu_1 \cup \mu_2 \vDash R\}.\end{aligned}$$

(4) If P is $\text{UNION}(P_1, P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$,
(5) If P is $\text{MINUS}(P_1, P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \setminus_M [\![P_2]\!]_G$,
(6) If P is $\text{FE}(P_1, P_2)$, then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G | [\![P_2\mu]\!]_G \neq \emptyset\}$,[11]

[11] Note that in our definition, the term $[\![P_2\mu]\!]_G$ in the evaluation of FE and FNE could be undefined as μ could map variables occurring in P_2 to blank

(7) If P is FNE (P_1, P_2), then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G | [\![P_2\mu]\!]_G = \emptyset\}$,
(8) If P is FILTER (P_1, R), then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G | \mu \vDash R\}$.

We define the evaluation of property paths over an RDF graph G as follows, based on Polleres (*31*). Let $s, o \in I \cup L \cup V$, x a fresh variable, and PP, PP_1 and PP_2 property paths. We define $S = var(\{s, o\})$[12], i.e., all variables found in s, o. We further introduce $\mu_{|S}$ which restricts a mapping μ to the set of variables S such that $dom(\mu_{|S}) = dom(\mu) \cap S$ and $\forall x \in S, x\mu_{|S} = x\mu$.

(1) $[\![\langle s, PP_1/PP_2, o\rangle]\!]_G = \{\mu_{|S} \mid \mu \in ([\![\langle s, PP_1, x\rangle]\!]_G \bowtie [\![\langle x, PP_2, o\rangle]\!]_G \}$
(2) $[\![\langle s, PP_1 | PP_2, o\rangle]\!]_G = [\![\langle s, PP_1, o\rangle]\!]_G \cup [\![\langle s, PP_2, o\rangle]\!]_G$
(3) $[\![\langle s, \hat{}PP, o\rangle]\!]_G = [\![\langle o, PP, s\rangle]\!]_G$
(4) $[\![\langle s, !(N, N'), o\rangle]\!]_G = [\![\langle s, !(N), o\rangle]\!]_G \cup [\![\langle o, !(N'), s\rangle]\!]_G$ where N represents the set of regular IRIs and N' the set of inverse IRIs.
(5) $[\![\langle s, !(N), o\rangle]\!]_G = \{\mu_{|S} \mid \langle s, x, o\rangle\mu \in G$ with $x\mu \notin N\}$
(6) $[\![\langle s, PP?, o\rangle]\!]_G = \{\mu \mid \mu \in [\![\langle s, PP, o\rangle]\!]_G\} \cup \{\mu \mid s\mu = o\mu, dom(\mu) = S, s\mu \in T_G\}$, where T_G the set of non-variable terms occurring in G.
(7) $[\![\langle s, PP^*, o\rangle]\!]_G = \{\mu \mid o\mu \in ALP(s\mu, PP, G)\}$
(8) $[\![\langle s, PP+, o\rangle]\!]_G = \{\mu_{|S} \mid \mu \in [\![\langle s, PP, x\rangle]\!]_G, o\mu \in ALP(x\mu, PP, G\rangle\}$

The $ALP(x, PP, G)$ function computes the transitive closure of PP starting from x, and is defined in the SPARQL 1.1 specification (*6*) and by Polleres (*31*). We note that ALP's transitive closure starting from x includes x itself.

Scoped variables. For our considerations later in this paper, we introduce the notion of *scope* with which we want to determine the domain of a solution mapping independently of the data graph it is evaluated on. In most cases, this domain is easy to determine. For basic graph patterns, the domain of the solution mapping coincides with the variables occurring in the pattern. There are two exceptions: For OPTIONAL and UNION the domain of their solution mapping always depends on the data graph the query is evaluated on and we only know which variables can possibly be in the solution mapping and which ones need to be in there for sure. We therefore define two concepts of scopes: the *maximal scope* sv and the *minimal scope* sw.

$sv : GP \to 2^V$ is defined as follows:

(1) If P is a BGP t, then $sv(P) = var(t)$,
(2) If $P = X(P_1, P_2)$, with $X \in \{\text{AND}, \text{UNION}\}$ then $sv(P) = sv(P_1) \cup sv(P_2)$,
(3) If $P = \text{OPT}(P_1, P_2, R)$, then $sv(P) = sv(P_1) \cup sv(P_2)$
(4) If $P = X(P_1, P_2)$, with $X \in \{\text{MINUS}, \text{FE}, \text{FNE}\}$ then $sv(P) = sv(P_1)$,
(5) If $P = \text{FILTER}\ (P_1, R)$, then $sv(P) = sv(P_1)$.

Note, that this function coincides with the scope as defined in the SPARQL specification (*6*, 18.2.1).

To determine the minimal domain of a solution mapping, we only have to slightly modify the previous definition. We define $sw : GP \to 2^V$ as:

(1) If P is a BGP t, then $sw(P) = var(t)$,
(2) If $P = \text{AND}(P_1, P_2)$, then $sw(P) = sw(P_1) \cup sw(P_2)$,
(3) If $P = \text{UNION}(P_1, P_2)$, then $sw(P) = sw(P_1) \cap sw(P_2)$,
(4) If $P = \text{OPT}(P_1, P_2, R)$, then $sw(P) = sw(P_1)$
(5) If $P = X(P_1, P_2)$, with $X \in \{\text{MINUS}, \text{FE}, \text{FNE}\}$ then $sw(P) = sw(P_1)$,
(6) If $P = \text{FILTER}\ (P_1, R)$, then $sw(P) = sw(P_1)$.

Note that the main difference between these two definitions is the treatment of UNION and OPTIONAL, where we take the union to get a maximal domain and the intersection and the first argument, respectively, to get the minimal domain.

Evaluation of SPARQL queries Aside from graph patterns discussed above, a SPARQL query also comes with a *query form*:

$$(n\ x)\ \text{WHERE}\ P$$

Where P is a graph pattern and the pair (n, x), called the query form, consists of a name $n \in \{\text{SELECT}, \text{CONSTRUCT}\}$, and argument x. If $n = \text{SELECT}$, this argument is a list of variables or $\star$. In that case, the query evaluation retrieves (parts of) the variable bindings μ given by $[\![P]\!]_G$, possibly considering multiple occurrences (multiset semantics). If $n = \text{CONSTRUCT}$, the argument x is the query template $Q \subset (I \cup L \cup V \cup B) \times (I \cup V) \times (I \cup L \cup V \cup B)$. In this case, the query evaluation returns instantiated versions of $Q\mu$, which replace each x occurring in Q by $\mu(x)$, and, in this case, also each blank node b by a blank node b_μ such that b_μ does not occur in G and $b_\mu \neq b'_\mu$ if $b \neq b'$ (i.e., different blank nodes b yield different b_μ).

With the current definition for CONSTRUCT queries, it can happen that μ leaves variables in the query template Q unbound. For instance, this is the case if Q contains "new" variables, that is if $var(Q) \setminus sv(P) \neq \emptyset$, or if Q contains variables from the optional clause (P_2) in an OPT pattern. In general, a SPARQL CONSTRUCT query leaves out any instantiated template triple that would be illegal in RDF (*6*). This includes triples with an unbound variable, as described above; or with an element in an illegal position, such as

nodes which we excluded. In that case we treat the blank nodes as constants. Here, we deviate from SPARQL 1.1 (*6*) to address the known issues related to the EXISTS pattern (*33*).

[12] We slightly abuse notation here as the var function is defined for graph patterns.

a literal in predicate position. In contrast, SiN3 will still produce triples with unbound variables, by replacing the latter by "witnesses" in the output, i.e., specially created blank nodes. Secondly, N3, as a superset of RDF, allows elements in positions that are illegal in RDF (e.g., literals as predicates), and SiN3 will also produce those triples. Hence, SiN3 deviates from SPARQL in this regard.

RDF-safety of `CONSTRUCT`. The discussion above highlights a corner case of `CONSTRUCT`: SPARQL returns an RDF graph and therefore omits any instantiated template triple that is illegal in RDF, whereas SiN3 will still keep such statements if they are legal in N3. In the remainder of this paper, whenever we claim equality of `CONSTRUCT` results between SPARQL and SiN3, we implicitly restrict attention to *RDF-safe* `CONSTRUCT` queries, i.e., queries (`CONSTRUCT` Q) `WHERE` P such that for every RDF graph G and every solution mapping $\mu \in [\![P]\!]_G$, the instantiated template $Q\mu$ is a *valid RDF graph* (in particular: no unbound variables remain in triple positions and no RDF-illegal term occurs in subject/predicate/object position). For RDF-unsafe queries, SiN3 may additionally output N3-legal statements that SPARQL intentionally drops.

5 Notation3

Notation3 Logic (N3) extends RDF with first-class lists, universal variables and graph terms, among others. Importantly, it offers rule-based reasoning in an open SW environment (*5*). As a SW language, N3 operates natively over the same primitives as RDF (IRIs, blank nodes, literals), thereby avoiding the "impedance mismatch" faced by other languages such as Datalog. In more detail, N3 incorporates the syntactic sugar of RDF 1.1 Turtle (*34*), including predicate/object lists and unlabeled blank nodes. Lists are first-class citizens with a suite of builtins for accessing and manipulating them. Graph terms are conjunctions of quoted statements to which metadata, such as provenance, may be attached. Rules offer scoped negation-as-failure and can reason over both local and online sources. For the full specification, see the W3C Community Group Report (*5*).

5.1 Syntax

The syntax of SPARQL and N3 is closely related (Section 2). We define N3 Graph Patterns (N3GP) as an extension of SPARQL Basic Graph Patterns (BGPs), i.e., $BGP \subset N3GP$. SPARQL BGP can thus be directly used within the premises and conclusions of N3 rules, with N3 allowing extra elements such as graph terms.

Given the sets I, B, L, and V, representing IRIs, Blank Nodes, RDF Literals, and Variables, respectively, we define the set $T_{N3} := AT \cup GT \cup LT$ of N3 terms:

- $AT := I \cup B \cup L \cup V$ is the set of *atomic terms*;
- $GT := \{\langle G \rangle \mid G \in N3GP\}$ is the set of *graph terms*;
- $LT := \{(t_1, \ldots, t_n) \mid n \geq 0,$ with elements $t_i \in T_{N3}\}$ is the set of *list terms*.

We call a tuple in $T_{N3} \times T_{N3} \times T_{N3}$ an *N3 triple*. An *N3 graph pattern* (N3GP) is a set of N3 triples.

Closed terms and graphs. We denote the sets of closed graph terms by GT_c and closed lists by LT_c, and define closed terms $T_c := I \cup L \cup GT_c \cup LT_c$. A graph term is closed if the term, and its possibly nested graph terms, contains no variables. A list is closed if it is the empty list, or it only includes elements from T_c.

A closed graph (also called a closed graph pattern) is a set of closed triples. Note that a closed graph does not allow any variables or blank nodes as subjects, predicates or objects. Graph terms in these positions can however contain blank nodes. From a semantic point of view, closed graphs are the N3 counterparts to ground graphs in RDF; they are variable-free and all their blank nodes are "safely" quantified within graph terms. For details, we refer to the specification (*5*).

5.2 Semantics

N3 semantics extends RDF simple entailment (*35*) to accommodate graph terms, lists, and logical operations (*36*). We present a simplified account that focuses on the aspects relevant to this paper, in particular the treatment of graph terms, mappings for variables and blank nodes, and the logical built-ins used for rule evaluation.

Base semantics

Interpretation of closed terms. N3's basic interpretation function $\mathfrak{I}$ maps closed terms into the domain of discourse $\Delta_\mathfrak{I}$. We ensure that the domain $\Delta_\mathfrak{I}$ also contains normalised syntactic representations of graph terms, and canonical representations of lists. $\mathfrak{I}$ maps graph terms and lists to these representations in $\Delta_\mathfrak{I}$. For all other closed terms, $\mathfrak{I}$ coincides with RDF's interpretation function I_{RDF}[13].

Mappings for variables and blank nodes. To interpret non-closed N3 graph patterns, we rely on mappings for variables and blank nodes . These mappings can be considered a combination of RDF's mapping function A (which maps blank nodes to the domain of discourse) and SPARQL variable mappings (Section 4.2).

We use mappings of the form

$$A : X \to \Delta_\mathfrak{I} \times T_c,$$

where $X \subseteq B$ or $X \subseteq V$, and T_c is the set of closed terms defined above. For any $x \in X$, $A(x) = (A_1(x), A_2(x))$, with $A_1(x)$ being a domain element from $\Delta_\mathfrak{I}$ and $A_2(x)$ being a closed term from T_c. We further require

$$\mathfrak{I}(A_2(x)) = A_1(x). \quad (1)$$

[13] We add an index here to avoid confusion with the set I of IRIs

The reason for the split is as follows: if a variable occurs in a graph term, we require $A_2(x)$ to ground the variable. After grounding, we obtain a closed graph term, which can then be interpreted by $\mathfrak{I}$ (see above). For variables outside of graph terms, we simply use $A_1(x)$ to directly map the variable into the domain of discourse. The condition from (1) ensures that if the variable grounded by A_2 is further interpreted (as in N3 semantics, Section 5.2), it refers to the same domain element that A_1 assigns to the variable.

Variables are universally quantified and blank nodes are existentially quantified. In most cases, the scope of variables is global, whereas the scope of blank nodes is local to the graph term in which they occur. The only exception here is formed by triples including the N3 predicates `log:includes` and `log:notIncludes`, abbreviated by "incl" and "notIncl", respectively, whose variables are subject to local scoping. We will discuss the details of this exception further below.

Truth of triples and graphs. We interpret relations using RDF's function *IEXT*. A triple $\langle t_1, t_2, t_3\rangle$ is true iff

$$(\mathfrak{I}(t_1), \mathfrak{I}(t_3)) \in \text{IEXT}(\mathfrak{I}(t_2)).$$

An N3 graph is true iff all its triples are true. If a graph G is true under a basic interpretation $\mathfrak{I}$, we write $\mathfrak{I} \models_b G$. Given a mapping A, we write $\mathfrak{I}[A]$ for the interpretation that uses A to interpret variables and blank nodes. Formally, for a term t we have:

$$\mathfrak{I}[A](t) = \begin{cases} A_1(t), \text{if } t \in V \cup B, \\ \mathfrak{I}(\langle G' A_2 \mid_V \rangle), \text{if } t = \langle G' \rangle \text{ is a graph term}, \\ \mathfrak{I}(t), \text{ else.} \end{cases}$$

$G'A_2$ is the result of applying the substitution A_2 (A_2 being the aforementioned function from variables or blank nodes to terms) on the graph G'. To ensure local scoping of blank nodes, the function A_2 is restricted to only variables ($A \mid_V$).

For an N3 graph pattern G, we say that $\mathfrak{I} \models G$ if for each mapping $A^V : V \to \Delta_\mathfrak{I} \times T_c$ there exists a mapping $A^B : B \to \Delta_\mathfrak{I} \times T_c$, both fulfilling requirement (1), such that for the combined mapping $A^V \cup A^B$, i.e. the mapping which applies A^V for elements of V and A^B for elements of B, the following holds: $\mathfrak{I}[A^V \cup A^B] \models G$.

N3 semantics

N3 models. Logical operations in N3 are expressed by built-in predicates, whose special meaning is not covered by the base semantics. For the logical built-in predicates, which we collect in a set LP, we define the *N3 semantics* as an extension of the base semantics.

An interpretation $\mathfrak{I}$ is an *N3 model* for a graph G, written $\mathfrak{I} \models_{n_3} G$, iff

1. $\mathfrak{I} \models_b G$,
2. for each triple $\langle s, p, o\rangle$ with $p \in LP$, the triple satisfies additional conditions that depend on the particular p.

In that case, we also say that the graph G is true under the N3 interpretation $\mathfrak{I}$. Below, we define the additional conditions for the predicates relevant to this paper.

Implication. In N3 rules are expressed using triples. For the predicate `log:implies` we have the arrow `=>` as syntactic sugar. In our abstract rules below, we abbreviate this with "impl" (or "$\rightarrow$" to emphasize the rule character of the triple). We define the following condition for the predicate:

$(s, o) \in \text{IEXT}(\mathfrak{I}(\text{impl}))$ if s or o is not a closed graph term or for $s = \langle G_1\rangle$ and $o = \langle G_2\rangle$ it holds that:

$$\text{if } \mathfrak{I} \models_{n_3} G_1 \text{ then } \mathfrak{I} \models_{n_3} G_2.$$

I.e., a triple with `impl` is satisfied when, under every grounding of the variables compatible with the base semantics, the truth of premise s entails the truth of conclusion o under the N3 interpretation $\mathfrak{I}$.

Local scoping. As mentioned before, the predicates `log:notIncludes` and `log:includes` are subject to local scoping. Consider a triple of the form $\langle s, \text{incl}, o\rangle$ or $\langle s, \text{notIncl}, o\rangle$ occurring in an N3 graph term $\langle G\rangle$ that serves as premise of a rule r (i.e., $r = \langle\langle G\rangle, \text{impl}, \langle F\rangle\rangle$). In case its object o is a graph term containing a variable $?x$ which does not occur anywhere else in G (outside of o), then $?x$ is treated as a locally scoped existential variable. For example, the triples $\langle\{\{\langle s, p, o\rangle\}, \text{incl}\{\langle s, c, ?x\rangle\}\}, \text{impl}, \langle\{\langle a, b, c\rangle\}\}\rangle$ and $\langle\{\{\langle s, p, o\rangle\}, \text{incl}\{\langle s, c, \_{:}\text{y}\rangle\}\}, \text{impl}, \langle\{\langle a, b, c\rangle\}\}\rangle$ have the same meaning. Having this peculiarity in mind, we can define the semantics of these predicates.

Inclusion. For predicate `log:includes`, abbreviated by "incl", we define:

$(s, o) \in \text{IEXT}(\mathfrak{I}(\text{incl}))$ if and only if $s = \langle G_1\rangle$ and $o = \langle G_2\rangle$ are closed graph terms and $G_1 \models_b G_2$.

I.e., a triple with `incl` is satisfied if, for every grounding of variables compatible with the base semantics, the graph from the term denoted by o is entailed by the same from s under the base semantics. A special case here is when the subject of the triple is left unspecified. Let F be the input graph: if we write $\langle \_, \text{incl}, o\rangle$[14], the subject is instantiated by the deductive closure $close(F)$, i.e., by a graph term with all triples that can be derived from F[15] by N3 entailment.

Negated inclusion. The predicate `log:notIncludes`, abbreviated by "notIncl", forms the counterpart of `log:includes`. Here, we put the condition:

$(s, o) \in \text{IEXT}(\mathfrak{I}(\text{notIncl}))$ if $s = \langle G_1\rangle$ and $o = \langle G_2\rangle$ are closed graph terms and $G_1 \not\models_b G_2$.

I.e., $\langle s, \text{notIncl}, o\rangle$ holds whenever, under the current N3 interpretation, there is no admissible grounding of the

[14] In concrete syntax, we use an arbitrary blank node instead of _.

[15] N3 does normally not allow infinite graph terms. Also, in N3, the input graph also includes the rules.

variables for which the graph from the term denoted by o is base-entailed by the same from s. When the subject is left unspecified and thus instantiated by the deductive closure $close(F)$, notIncl expresses that the object graph term cannot be derived from the input graph F (and its rules) under N3 entailment. This yields a scoped form of negation-as-failure over graph terms and possibly the input graph.

Shorthands and covered logical built-ins. Throughout this paper, we use the shorthands for N3 `log` built-ins summarized in Table 2. The equality predicates `eq` and `noteq` are used by the translation and runtime rules; we adopt their standard meaning from the N3 specification (*5*).

Short	Prefixed name
`impl`	`log:implies`
`incl`	`log:includes`
`notIncl`	`log:notIncludes`
`eq`	`log:equalTo`
`noteq`	`log:notEqualTo`

Table 2. Shorthand glossary for the N3 logical built-ins used in this paper.

5.3 Forward and backward reasoning

Implementations realize the above semantics using different execution strategies; some reasoners (notably eye (*14*)) even allow the user to specify how rules are applied. *Forward* (bottom-up) reasoning applies rules to initial and inferred facts to infer new facts, iterating until no more new conclusions can be inferred (saturation). *Backward* (top-down) reasoning starts from a goal (e.g., `?x a :Person`), seeks rules whose conclusions might satisfy it, and then checks their premises against facts or other rules' conclusions, stopping when all premises are resolved. Both strategies are sound with respect to N3 entailment; provided they terminate and are complete for the considered rule fragment, they also yield the same set of entailed conclusions. Both strategies can be considered equivalent in expressivity, while they are clearly operationally different. For instance, in practice, backward reasoning can avoid exploring large parts of the search space, as illustrated in our evaluation (Section 7). For the rules shown in this paper, we use an arrow $\rightarrow$ from left to right to indicate that we expect a rule $A \rightarrow B$ to be evaluated in a forward chaining manner. We write the same rule $B \leftarrow A$ in the reverse order to indicate backward chaining.

6 From SPARQL to N3

In this section, we define the mapping between SPARQL and N3. Provided with a SPARQL query

$$(n\ x) \text{ WHERE } P$$

Our translated rule is given by

$$m(P) \rightarrow h((n\ x), P)$$

Where we define a premise mapping m from the SPARQL graph pattern P to an N3 graph pattern as the rule premise or body (Section 6.1); and a conclusion mapping h from the SPARQL query to an N3 graph pattern as the rule conclusion or head (Section 6.2).

With this setup, each SPARQL query is represented by one single N3 rule, thus adhering to our one query–one rule principle. However, adhering to this principle required the direct translation of SPARQL concepts to equivalent N3 constructs, which were lacking for the SPARQL UNION and recursive property paths $*, +$. In a logical rule language, these concepts are typically, if not always, implemented using multiple rules: most rule languages do not allow disjunction in the premise, and recursion, even the limited type required by recursive property paths, always depends on multiple rule applications. We thus manually implement these as N3-native constructs using an extra "runtime" ruleset. That is, when those SPARQL constructs are utilized, the translated queries $m(P) \rightarrow h((n\ x), P)$ need to be combined with this fixed runtime ruleset. We elaborate on our runtime ruleset in Section 6.3.

6.1 From SPARQL graph patterns to N3 premises

We explain our translation in two steps: we first introduce the function for translating property paths, and then discuss the translation of query patterns.

Property Paths We introduce a function pp which maps triples $\langle s, PP, o\rangle$, where PP is a SPARQL property expression, to a set of N3 triples. The function is defined as follows:

- $pp(\langle s, p, o\rangle) = \{\langle s, p, o\rangle\}$, if p is an IRI or variable,
- $pp(\langle s, P_1/P_2, o\rangle) = pp(\langle s, P_1, x\rangle) \cup pp(\langle x, P_2, o\rangle)$, with $x \in V$ a "fresh" variable,
- $pp(\langle s, P_1|P_2, o\rangle) =$ $\{\langle\ pp(\langle s, P_1, o\rangle), \text{union}, pp(\langle s, P_2, o\rangle)\ \rangle\}$,
- $pp(\langle s, \hat{}P, o\rangle) = pp(\langle o, P, s\rangle)$,
- $pp(\langle s, !(p_1|\ldots|p_i|\hat{}p_{i+1}|\ldots|\hat{}p_n), o\rangle) =$ $\{\langle\ N_1, \text{union}, N_2\ \rangle\}$, with $N_1 = \{\langle s, x, o\rangle\} \cup \{\langle x, \text{noteq}, p_k\rangle \mid 1 \leq k \leq i\}$ and $N_2 = \{\langle o, x, s\rangle\} \cup \{\langle x, \text{noteq}, p_k\rangle \mid i < k \leq n\}$, and $x \in V$ a "fresh" variable,
- $pp(\langle s, P?, o\rangle) =$ $\{\langle\{\langle\ pp(\langle s, P, o\rangle\}, \text{union}, \{\langle\{\langle y, x, s\rangle\}\rangle, \langle s, \text{eq}, o\rangle\}\rangle\}$ with $x, y \in V$ "fresh" variables,
- $pp(\langle s, P^+, o\rangle) = \{\langle s, pp_{N_3}(P^+), o\rangle\}$,

- $pp(\langle s, P^*, o\rangle) =$
  $\{\langle \{\langle s, pp_{N_3}(P^+), o\rangle\}, \text{union},$
  $\{\langle\{\langle y, x, s\rangle\}, \text{union}, \{\langle s, y, x\rangle\}\rangle, \langle s, \text{eq}, o\rangle\}\rangle\}\ \rangle\}.$

The definition makes use of the N3 "union" predicate, which we implemented as part of the runtime rules to express SPARQL UNION (Section 6.3). Similarly, we refer to a function pp_{N_3} that translates the recursive property path P^+ to an N3 list predicate, which is then handled by our runtime rules (Section 6.3). We define the function pp_{N_3} as follows[16]:

$$
\begin{aligned}
pp_{N_3}(p) &= p \quad \text{for an IRI } p,\\
pp_{N_3}(P) &= (\text{"" } pp_{N_3}(P)),\\
pp_{N_3}(P_1/P_2) &= (pp_{N_3}(P_1)\ \text{"/"}\ pp_{N_3}(P_2)),\\
pp_{N_3}(P_1|P_2) &= (pp_{N_3}(P_1)\ \text{"|"}\ pp_{N_3}(P_2)),\\
pp_{N_3}(P^*) &= (pp_{N_3}(P)\ \text{"*"}),\\
pp_{N_3}(P^+) &= (pp_{N_3}(P)\ \text{"+"}),\\
pp_{N_3}(!N) &= (\text{"!"}\ (pp_{N_3}(e_i) \mid e_i \in N)).
\end{aligned}
$$

Query Patterns For our translation of query patterns, we syntactically restrict the allowed SPARQL queries. We first introduce the restriction and the need for it. Next, we define the translation and the auxiliary functions it relies on.

Well-Designed Graph patterns. For the remainder of this paper, we assume all SPARQL patterns to be well-designed. The concept was originally introduced by Perez et al. (*30*). We use the extension of the definition provided by Polleres (*15*) which allows queries which are not in UNION normal form. We further extended the definition to also cope with the query types MINUS, FE and FNE which were not part of the SPARQL standard when the concept was introduced. We first formally introduce the restriction and then explain the reason for the restriction.

A SPARQL graph pattern $P \in GP$ is *well-designed* if, for all its sub queries $P' \in sub(P)$, the following holds: If P' is of the form $\text{OPT}(P_1, P_2, R)$ or $\text{UNION}(P_1, P_2)$ then for all queries $Q \in pd(P, P') \setminus sup(P', P)$ it holds that $sv(Q) \cap sv(P') \subseteq sw(P')$. Informally speaking, the definition means that if a variable $?x$ occurs in a query pattern P and is in the maximum scope of two subpatterns, of which one is either a OPT or a UNION pattern, then it is also in the minimum scope of this query.

To explain the need for the requirement, we first take a closer look at an example. The query in listing 7 has an OPTIONAL query pattern $Q = \text{OPT}(Q_1, Q_2, \top)$ (lines 3-8) with arguments $Q_1 = \{\langle s1, p, ?v\rangle\}$ (line 4) and $Q_2 = \text{OPT}(\{\langle s2, q, ?w\rangle\}, \{\langle s3, p, ?v\rangle\}, \top)$ (lines 5-6). Q_2 is again an OPTIONAL pattern. The second argument ($\{\langle s3, p, ?v\rangle\}$) of Q_2 shares the variable $?v$ with Q_1 but not with the first argument of Q_2. The query is thus not well designed.

If we evaluate the query on the data in listing 8 we first consider Q_1 and Q_2 separately and retrieve the solution mappings $[\![Q_1]\!]_G = \{\{(?v, 1)\}\}$ and $[\![Q_2]\!]_G = \{\{(?v, 3), (?w, 2)\}\}$. As $\mu_1 = \{(?v, 1)\}$ is not compatible to the solution mapping $\mu_2 = \{(?v, 3), (?w, 2)\}$, since the mappings assign different values to the variable $?v$, we get μ_1 as evaluation result of the overall query.

```
1 PREFIX : <http://example/>
2 SELECT *
3     {
4       :s1 :p ?v .
5       OPTIONAL { :s2 :q ?w .
6         OPTIONAL { :s3 :p ?v }
7       }
8     }
```

Listing 7: Nested OPTIONAL query.

```
1 @prefix : <http://example/> .
2 :s1 :p 1 .
3 :s2 :q 2 .
4 :s3 :p 3 .
```

Listing 8: Example graph.

In our implementation, we aim for translating one query pattern Q to one rule premise (one query–one rule principle). In a single rule premise, it is not possible to assign different values to the same variable under model-theoretic semantics (i.e., dealing with patterns that are instantiated as a whole). This is in contrast to the SPARQL semantics illustrated above, which deal with separate variable mappings. This is a clear illustration of where the mismatch between these two semantics leads to significant—and perhaps unexpected—complexity when translating certain constructs. An implementation of the unrestricted OPTIONAL pattern would need to rely on auxiliary runtime rules, possibly combined with variable renaming.[17] We thus impose the well-designedness restriction, which ensures that the variables occurring in translated rules can be bound to only one value.

Note that our definition of well-designedness differs from Perez et al.'s and Polleres' definition in one aspect: we mainly consider the maximum scope of sub patterns. As our notion of sub patterns of a query is rather broad, this difference mainly influences variables occurring in FILTER expressions. We allow these to coincide with variables occurring under OPTIONAL or UNION patterns in the SPARQL query and rename isolated FILTER variables during our translation to avoid unintended unification of variables.

MINUS *in well designed patterns.* Similar to the possible difficulties arising from OPTIONAL patterns, we encounter

[16] All property paths need to be supported as they may be nested in P^+.

[17] Prior work (*13*) provided such rules, but used N3 built-in predicates whose semantics are not formally defined yet, which hindered a formal verification of the translation.

difficulties with MINUS patterns. These are caused by the fact that, when evaluating a query, the domain of a solution mapping can depend on the data graph. For example,

$$\text{MINUS}(\{\langle s, p1, ?x\rangle\}, \text{OPT}(\{\langle s, p2, ?x\rangle\}, \{\langle s, p3, ?y\rangle\}, \top))$$

has this property. For the query

$$Q_2 = \text{OPT}(\{\langle s, p2, ?x\rangle\}, \{\langle s, p3, ?y\rangle\}, \top),$$

the maximal scope satisfies $sv(Q_2) = \{?x, ?y\}$, whereas the minimal scope is $sw(Q_2) = \{?x\}$. Without knowing the data graph, we therefore cannot determine whether the domain of a solution mapping for Q_2 overlaps with that of the solution mapping for $Q_1 = \{\langle s, p1, ?x\rangle\}$, where $sv(Q_1) = sw(Q_1) = dom(Q_1) = \{?x\}$. Fortunately, the query above is excluded from our translation because it is not well designed, and we show below that this condition rules out the problem.

Lemma 1. *Let P be a well designed graph pattern and $Q = MINUS(Q_1, Q_2) \in sub(P)$ a sub pattern of P. Then $sw(Q_1) \cap sw(Q_2) = sv(Q_1) \cap sv(Q_2)$.*

Proof. We first note that for all graph patterns $Q_1, Q_2 \in GP$: $sw(Q_1) \cap sw(Q_2) \subseteq sv(Q_1) \cap sv(Q_2)$. It remains to show that $sv(Q_1) \cap sv(Q_2) \subseteq sw(Q_1) \cap sw(Q_2)$

Let us assume that there exists an $x \in sv(Q_1) \cap sv(Q_2)$ with $x \notin sw(Q_1) \cap sw(Q_2)$, then there must be sub pattern $S \in sub(Q)$ of the form $S = \text{OPT}(S_1, S_2, R)$ or $S = \text{UNION}(S_1, S_2)$ such that $x \in sv(S)$ but $x \notin sw(S)$. As x occurs in the intersection of the maximum scopes of two sub patterns of Q, namely Q_1 and Q_2, we know that there exists a sub pattern $Q_i \in pd(P, S) \setminus sup(S, P)$ such that $x \in sv(Q_i) \cap sv(S)$, but because the well designedness of P, that means that $x \in sw(S)$ which leads to a contradiction.

From the above we can conclude:[18]

Corollary 2. *Let G be an RDF graph, $P \in GP$ be a well-designed graph pattern and $Q = MINUS(Q_1, Q_2) \in sub(P)$ a sub query of P. Then for all solution mappings $\mu_1 \in [\![Q_1]\!]_G$ and $\mu_2 \in [\![Q_2]\!]_G$ it holds that $dom(\mu_1) \cap dom(\mu_2) = sv(Q_1) \cap sv(Q_2)$.*

Variable renaming. Some query patterns, such as for example the FILTER or the MINUS, rely on a renaming of variables. We illustrate that with an example. Consider the nested MINUS pattern in listing 9. We evaluate the query on the example graph $G = \{\langle s, p1, 1\rangle, \langle s, p2, 2\rangle, \langle s, p3, 3\rangle\}$. We start with the nested MINUS query $Q_2 = \text{MINUS}(\{\langle ?x, p2, ?z\rangle\}, \langle ?x, p3, ?y\rangle\})$ (lines 4–6). For the subpattern $Q_{2,1} = \{\langle ?x, p2, ?z\rangle\}$ we retrieve a solution mapping $\mu_{2,1} = \{(?x, s), (?z, 2)\}$ and for $Q_{2,2} = \{\langle ?x, p3, ?y\rangle\}$ we get $\mu_{2,2} = \{(?x, s), (?y, 3)\}$. The domains of these solution mappings share a variable, namely $?x$, and they are compatible, we thus get

```
1 SELECT * WHERE
2     {
3         ?x :p1  ?y .
4         MINUS { ?x :p2 ?z.
5             MINUS {?x :p3 ?y}
6         }
7     }
```

Listing 9: Example of a nested MINUS query

$[\![Q_2]\!]_G = \emptyset$. Together with the evaluation of the pattern $Q_1 = \{\langle ?x, p1, ?y\rangle\}$ which yields the solution mapping $\mu_1 = \{(?x, s), (?y, 1)\}$ we get for our query Q from the listing that $[\![Q]\!]_G = \{\mu_1\}$. We again assigned two different values to the variable $?y$. While this is perfectly possible in the SPARQL execution semantics, when dealing with instantiated patterns (i.e., using model-theoretic semantics), one cannot assign two different values to the same variable in a single rule premise. Hence, we make the same note as before (*Well-Designed Graph Patterns*) on the mismatch between the semantics of query and rule languages leading to increased complexity.

In this case, we do not need to further restrict the query pattern but can directly rely on renaming. As we deal with well-designed query patterns, Corollary 2 ensures that we know, at translation time, which variables are shared between the domains of the solution mappings of the first and second arguments of a MINUS. All other variables need to be renamed during translation. The query pattern from our example would be translated into the rule premise:

$$\{\langle ?x, p1, ?y\rangle, \langle \_ \text{ notIncl}, \{\langle ?x, p2, \textbf{?z'}\rangle, \langle \_, \text{notIncl}, \{\langle ?x, p3, \textbf{?y'}\rangle\}\rangle\}\rangle\}$$

To make such a renaming possible, our translation relies on the relabeling function $rl : N3GP \times 2^V \to N3GP$ where $rl(P, X)$ is the graph P' we obtain by replacing all variables $var(P) \setminus X$ occurring in P by "fresh" ones, that is, by a variable neither occurring in P nor in any of its super patterns in case P is a sub query of a bigger query we need to translate. We have, for example, $rl(\{\langle ?x, p3, ?y\rangle\}, \{?x\}) = \{\langle ?x, p3, ?y'\rangle\}$. The relabeling function is needed for MINUS, FILTER, FNE, and FE.

Filter translation. To handle filter expressions, we use the function ft which maps a SPARQL filter condition R to its corresponding N3 pattern G_R. We note that N3 supports logical, string, numerical, etc. operations through builtin predicates used within N3 triples; SPARQL instead supports these operations via functions or expressions inside the FILTER clause (Section 2). For instance, SPARQL's

[18] Note that prior work (*13*) silently relies on an even stronger assumption, namely that independently of that data graph G it holds for all $\mu \in [\![Q]\!]_G$ that $dom(\mu) = sv(Q)$.

builtin `isLiteral(?x)` checks whether the value that the variable `?x` is mapped to during evaluation is a literal. In N3, this is instead expressed using the triple `?x log:rawType rdfs:Literal.` Despite this difference in syntax, the translation between filter conditions and N3 triples is rather straightforward; most SPARQL filter functions have direct N3 counterparts (i.e., builtin predicates), whereas some others rely on the application of several N3 builtin predicates. We thus do not further detail ft here.

Mapping m *from SPARQL to N3 Graph Patterns.* Our mapping relies on the maximum scope function sv, the relabeling function rl, and the filter translation ft which have all been introduced above.

Let P be a graph pattern. We define the premise mapping $m : GP \rightarrow N3GP$ recursively as follows:

- if P is a BGP Q, then $m(P) = \bigcup_{t \in Q} pp(t)$,
- if P is AND(P_1, P_2), then $m(P) = m(P_1) \cup m(P_2)$,
- if P is OPT(P_1, P_2, R), then

$$\begin{aligned} m(P) = \{&\langle \langle m(P_1) \cup m(P_2) \cup \overline{ft}(R) \rangle, \\ &\text{union}, \\ &\langle m(P_1) \cup \{\langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R) \rangle\} \rangle \rangle\}, \end{aligned}$$

  with $\overline{ft}(R) = rl(ft(R), sv(P_1) \cup sv(P_2))$,
- if P is UNION(P_1, P_2), then

$$m(P) = \{\langle \langle m(P_1) \rangle, \text{union}, \langle m(P_2) \rangle \rangle\},$$

- if P is MINUS(P_1, P_2), then. If $sv(P_2) \cap sv(P_1) = \emptyset$, then $m(P) = m(P_1)$. Otherwise,

$$\begin{aligned} m(P) = m(P_1) \cup \{&\langle \_, \text{notIncl}, \\ &\langle rl(m(P_2), sv(P_1)) \rangle \rangle\}. \end{aligned}$$

- if P is FNE(P_1, P_2), then

$$\begin{aligned} m(P) = m(P_1) \cup \{&\langle \_, \text{notIncl}, \\ &\langle rl(m(P_2), sv(P_1)) \rangle \rangle\}, \end{aligned}$$

- if P is FE(P_1, P_2), then

$$\begin{aligned} m(P) = m(P_1) \cup \{&\langle \_, \text{incl}, \\ &\langle rl(m(P_2), sv(P_1)) \rangle \rangle\}, \end{aligned}$$

- if P is FILTER(P_1, R), then

$$m(P) = m(P_1) \cup rl(ft(R), sv(P_1)).$$

Similar to before (Section 6.1), the translation simply converts UNION patterns to corresponding N3 triples with the "union" predicate, which is implemented by our runtime rules (Section 6.3). For the MINUS we can simply disregard the second operand P_2 in case the scopes $sv(P_1)$ and $sv(P_2)$ are disjoint. Else, we need to relabel all variables in P_2, *except* for those that also occur in the scope of P_1.

Below, we prove that the defined mapping m works correctly. We do that for ground RDF graphs (i.e., the data graph) which do not contain any builtin predicate of N3. We call these graphs proper RDF graphs. We limit our consideration to ground RDF graphs to avoid possible difficulties caused by local blank node scoping. As the filter translation function ft is beyond the scope of this paper and the predicate "union" will only be discussed in Section 6.3, we rely on the following assumptions:

Assumption 1 Given an RDF ground graph G, the set of run time rules for "union" $\mathcal{R}$ (Section 6.3) and a mapping $\mu : X \rightarrow I \cup L$. Then the following holds for each query pattern $P = \text{FILTER}(P, R)$[19]:

$$\mu \in [\![P]\!]_G \quad \text{iff} \quad \begin{array}{l} \mu \text{ is a minimal mapping such that:} \\ G \cup \mathcal{R} \models_{n_3} m(P)\mu. \end{array}$$

Assumption 2 Given an RDF ground graph G, the set of run time rules for "union" $\mathcal{R}$ (Section 6.3) and two N3 graph patterns P_1 and P_2, then:

$$\begin{array}{c} G \cup \mathcal{R} \models_{n_3} \langle \langle P_1 \rangle, \text{union}, \langle P_2 \rangle \rangle \\ \text{iff} \\ G \cup \mathcal{R} \models_{n_3} P_1 \text{ or } G \cup \mathcal{R} \models_{n_3} P_2 \end{array}$$

As with the above, only one of the two arguments needs to be true, the other argument can be arbitrarily. For our purpose of simulating SPARQL queries and retrieve their solution mappings, that is problematic because we for example have that $\{\langle s, p, o \rangle\} \cup \mathcal{R} \models_{n_3} \langle \langle s, p, o \rangle, \text{union}, \langle s, p, ?x \rangle \rangle$ which can lead to triples with arbitrary bindings being true. In practice, this is not a problem because an N3 reasoner will only produce instances for triples which do not hold universally, if, for example, $\langle s, p, o \rangle, \text{union}, \langle s, p, ?x \rangle \rangle$ holds, then it will not produce $\langle s, p, o \rangle, \text{union}, \langle s, p, 27 \rangle \rangle$. To mimic similar behavior for our proof we use *minimal* mappings. Here, we remind the reader, that we note mappings as sets of pairs. A mapping μ is a minimal mapping fulfilling a condition if the condition does not hold for any mapping $\mu' \subsetneq \mu$.

Before proving that the partly instantiated premise of a translated rule is true for all solution mappings, we want to remind the reader that "incl" and "notIncl" are subject to local scoping (Section 5.2). To cope with that we assume that in contexts where a variable is locally scoped, this variable is not affected by a substitution. For example, given that the pattern $P = \{\langle s, p, ?x \rangle, \langle \_, \text{notIncl}, \langle ?x, q, ?y \rangle \rangle\}$ and the substitution $\sigma = ?x/o, ?y/g$, $P\sigma = \{\langle s, p, o \rangle, \langle \_, \text{notIncl}, \langle o, q, ?y \rangle \rangle\}$ and not $\{\langle s, p, o \rangle, \langle \_, \text{notIncl}, \langle o, q, g \rangle \rangle\}$, as the variable $?y$ only occurs in the argument of "notIncl".

Under the above assumptions we show:

[19] The n3-entailment below is assumed to also cover the built-in functions used in the filter translations.

Lemma 3. *Given the runtime rules $\mathcal{R}$ for the predicate "*union*", a proper RDF graph G, a well designed SPARQL graph pattern P which does not contain property path expressions, and a mapping $\mu : X \to I \cup L$, then the following holds:*

$$\mu \in [\![P]\!]_G \quad \textit{iff} \quad \mu \textit{ is a minimal mapping such that: } G \cup \mathcal{R} \models_{n_3} m(P)\mu.$$

Proof. We use induction over the structure of P.

(1) *Let P be a BGP:*

"⇒" Let $\mu \in [\![P]\!]_G$ and let $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$. As P is a BGP, we have $m(P) = P$ and $P\mu \subseteq G$, therefore $\mathfrak{I} \models_{n_3} m(P)\mu$. As $dom(\mu) = var(P)$ in this case, μ is minimal.

"⇐" Let $G \cup \mathcal{R} \models_{n3} m(P)\mu$. As G and $P\mu$ are proper RDF graphs, $P\mu$ cannot be derived by applying rules of $\mathcal{R}$ on G. $P\mu$ must thus be an instance of G. We derive that $P\mu \subseteq G$ and thereby, as μ is minimal, $\mu \in [\![P]\!]_G$.

(2) *Let P be of the form* AND(P_1, P_2)*:*

"⇒" Let $\mu \in [\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$, That is $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in [\![P_1]\!]_G$ and $\mu_2 \in [\![P_2]\!]_G$.By the induction hypothesis that means μ_1 and μ_2 are minimal mappings such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu_1$ and $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu_2$. This implies $G \cup \mathcal{R} \models_{n_3} m(P)\mu$.

"⇐" Let μ be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P)\mu$. Then $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$ and $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu$. We can restrict the mappings such that $\mu_1 \subseteq \mu$ and $\mu_2 \subseteq \mu$ are minimal mappings such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu_1$ and $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu_2$. Because of the minimality of μ it must hold that $\mu = \mu_1 \cup \mu_2$. By the induction hypothesis we get $\mu_1 \in [\![P_1]\!]_G$ and $\mu_2 \in [\![P_2]\!]_G$ and thus $\mu \in [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.

(3) *Let P be of the form* MINUS(P_1, P_2)*:*

"⇒" Let $\mu \in [\![P]\!]_G = [\![P_1]\!]_G \setminus_M [\![P_2]\!]_G$. Then $\mu \in [\![P_1]\!]_G$ and for all solution mappings $\mu' \in [\![P_2]\!]_G$ we either have that $\text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset$ or $\mu \not\sim \mu'$. We consider these two cases separately:

- If $\text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset$ for all solution mappings $\mu' \in [\![P_2]\!]_G$, then, by Corollary 2, $sv(P_1) \cap sv(P_2) = \emptyset$, and we have that $m(P) = m(P_1)$. As $\mu \in [\![P_1]\!]_G$, we get by the induction hypothesis that μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$.
- Let $\mu \not\sim \mu'$ for all solution mappings $\mu' \in [\![P_2]\!]_G$. Let $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$. To show that $\mathfrak{I} \models_{n_3} (m(P_1) \cup \{\langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\})\mu$, we show that $\mathfrak{I} \models_{n_3} m(P_1)\mu$ and $\mathfrak{I} \models_{n_3} \{\langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_2))\rangle\rangle\}\mu$. We can treat these two parts separately as scoping of blank nodes and isolated variables in the graph term $\langle rl(m(P_2), sv(P_2))\rangle$ is local (Section 5.2). We know by the induction hypothesis that μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$. We can thus not find smaller mapping $\mu_{min} \subsetneq \mu$ such that $G \cup \mathcal{R} \models_{n_3} m(P)\mu_{min}$. It remains to show that $\mathfrak{I} \models_{n_3} \{\langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\}\mu$. We do that by contradiction assuming that $close(G \cup \mathcal{R}) \models_b rl(m(P_2), sv(P_1))\mu$ which implies $G \cup \mathcal{R} \models_{N3} rl(m(P_2), sv(P_1))\mu$. As $rl(m(P_2), sv(P_1))$ is a version of $m(P_2)$ with renamed variables, there exists a mapping $\sigma_{rl} : vars(P_2) \setminus sv(P_1) \to V$ such that $rl(m(P_2), sv(P_1)) = m(P_2)\sigma_{rl}$ and thus $G \cup \mathcal{R} \models_{n_3} m(P_2)\sigma_{rl}\mu$. If this mapping is not minimal, we take the minimal mapping $\mu'' \in \sigma_{rl}\mu$ such that $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu''$. By the induction hypothesis, this means that $\mu'' \in [\![P_2]\!]_G$ (note that the domains of μ and σ_{rl} were disjoint, which makes the union and its subsets a mapping). By construction, we furthermore have $\mu \sim \mu''$ which leads to a contradiction.

"⇐" Let $\mu : X \to I \cup L$ be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P)\mu$. We note that in this case If $sv(P_2) \cap sv(P_1) = \emptyset$, then $m(P) = m(P_1)$. Otherwise,

$$m(P) = m(P_1) \cup \{\langle \_, \text{notIncl}, rl(m(P_2), sv(P_1))\rangle\}.$$

We again consider the two cases separately:

- Let $sv(P_1) \cap sv(P_2) = \emptyset$, then $m(P) = m(P_1)$ and μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$. We get by the induction hypothesis that $\mu \in [\![P_1]\!]_G$. The claim follows by Corollary 2, as $dom(\mu) \cap dom(\mu') = sv(P_1) \cap sv(P_2) = \emptyset$ for all solution mappings $\mu' \in [\![P_2]\!]_G$.
- If $sv(P_1) \cap sv(P_2) \neq \emptyset$, then $m(P) = m(P_1) \cup \{\langle \_, \text{notIncl}, rl(m(P_2), sv(P_2))\rangle\}$. We have to show that $\mu \in [\![P_1]\!]_G \setminus_M [\![P_2]\!]_G$ which in this case means (Corollary 2) that for all solution mappings $\mu' \in [\![P_2]\!]_G$ we have $\mu \not\sim \mu'$. We do that by contradiction and assume that there exists a mapping $\mu' \in [\![P_2]\!]_G$ with $\mu \sim \mu'$. By the induction hypothesis, it holds that $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu'$. Let $\sigma_{rl} : vars(P_2) \setminus sv(P_1) \to V$ be the mapping such that $m(P_2)\sigma_{rl} = rl(m(P_2), sv(P_1))$. As σ_{rl} is injective and only assigns new variables, we get $m(P_2) = rl(m(P_2), sv(P_1))\sigma_{rl}^{-1}$ and thereby $G \cup \mathcal{R} \models_{n_3} rl(m(P_2), sv(P_1))\sigma_{rl}^{-1}\mu'$ which implies $close(G \cup \mathcal{R}) \models_b rl(m(P_2), sv(P_1))\sigma_{rl}^{-1}\mu'$. As $\mu \sim \mu'$, that is the same as $close(G \cup \mathcal{R}) \models_b (rl(m(P_2), sv(P_1))\mu)(\sigma_{rl}^{-1}\mu')$. That is, there exists grounding of variables such that the closure of $G \cup \mathcal{R}$ b-entails an instance of $rl(m(P_2), sv(P_1))\mu$. This contradicts $G \cup \mathcal{R} \models_{n_3} m(P)\mu$.

(4) *Let P be of the form* UNION(P_1, P_2)*:*

"⇒" Let $\mu \in [\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$. By induction hypothesis this means that μ is a minimal mapping

fulfilling $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$ or a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu$. By Assumption 2 this means that μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P)\mu$.

"⇐" Let μ be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} \langle\langle m(P_1)\rangle, \text{union}, \langle m(P_2)\rangle\rangle\mu$. By Assumption 2, this means that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$ or $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu$ which, by induction hypothesis, leads to $\mu \in [\![P_1]\!]_G \cup [\![P_2]\!]_G$.

(5) *Let P be of the form* $\text{OPT}(P_1, P_2, R)$*:*

"⇒" Let $\mu \in [\![\text{OPT}(P_1, P_2, R)]\!]_G$. Then either (a) $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G$, $\mu_1 \sim \mu_2$ and $\mu_1 \cup \mu_2 \vDash R$, or (b) $\mu = \mu_1 \in [\![P_1]\!]_G$ and no $\mu_2 \in [\![P_2]\!]_G$ as above exists. We consider the two cases separately.
(a) Let $\mu = \mu_1 \cup \mu_2$ as above. As $\mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$ and $\mu_1 \sim \mu_2$, we have that $\mu \in [\![\text{AND}(P_1, P_2)]\!]$ and, with the filter condition, thus in $\text{FILTER}(\text{AND}(P_1, P_2), R)$. By the induction Hypothesis and Assumption 1, this means that μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} (m(P_1) \cup m(P_2)) \cup \overline{ft}(R))\mu$.
(b) Let $\mu \in [\![P_1]\!]_G$ and there exists no mapping $\mu' \in [\![P_2]\!]_G$ such that $\mu \sim \mu'$ and $\mu \cup \mu' \models R$. By induction hypothesis, $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$. We show that $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R)\rangle\mu$ by contradiction. We assume that $close(G \cup \mathcal{R}) \models_b (m(P_2) \cup \overline{ft}(R))\mu$. That is, given a model $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$ there exists a mapping $A : var(P_2 \cup \overline{ft}(R)) \setminus dom(\mu) \to \Delta_{\mathfrak{I}} \times T_G$ such that $\mathfrak{I}[A] \models_{n_3} (m(P_2) \cup \overline{ft}(R))\mu$. This implies for $A_2 : var(m(P_2) \cup \overline{ft}(R)) \setminus dom(\mu) \to T_G$ that $\mathfrak{I} \models (m(P_2) \cup \overline{ft}(R))(\mu \cup A_2)$. From this mapping, we can obtain a minimal mapping $\sigma \subseteq \mu \cup A_2$ fulfilling the above condition. By induction hypothesis we then have $\sigma \in [\![P_2]\!]_G$ and $\mu \cup \sigma \vDash R$. As $\mu \sim \sigma$, this leads to a contradiction.

"⇐" Let μ be a minimal mapping such that $G \cup \mathcal{R} \models m(P)\mu$. As

$$m(P) = \{\langle\langle m(P_1) \cup m(P_2) \cup \overline{ft}(R)\rangle, \text{union}, \langle m(P_1) \cup \{\langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R)\rangle\}\rangle\rangle\},$$

we know by Assumption 2 that
(a) $G \cup \mathcal{R} \models_{n_3} (m(P_1) \cup m(P_2) \cup \overline{ft}(R))\mu$
or
(b) $G \cup \mathcal{R} \models_{n_3} m(P_1) \cup \{\langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R)\rangle\}\mu$
We treat these cases separately:
(a) Let μ Be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} (m(P_1) \cup m(P_2) \cup \overline{ft}(R))\mu$. By induction hypothesis and Assumption 1, $\mu \in [\![\text{FILTER}(\text{AND}(P_1, P_2), R)]\!]_G$. That is, $\mu \in [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$ and $\mu \vDash R$. By definition, this means that $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$ and $\mu_1 \sim \mu_2$. We get $\mu \in [\![\text{OPT}(P_1, P_2, R)]\!]_G$.
(b) Let μ be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} (m(P_1) \cup \{\langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R)\rangle\})\mu$. As the arguments of notIncl are scoped locally, that means that μ is also a minimal mapping such that $G \cup \mathcal{R} \models m(P_1)\mu$ and we get $\mu \in [\![P_1]\!]_G$. We prove that there exists no mapping $\mu' \in [\![P_2]\!]_G$, such that $\mu \sim \mu'$ and $\mu \cup \mu' \vDash R$ by contradiction. Let $\mu' \in [\![P_2]\!]_G$ be a mapping such that $\mu \sim \mu'$ and $\mu \cup \mu' \vDash R$. Then, by induction hypothesis, $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu'$ which, because of the compatibility, leads to $\mu \cup \mu' \in [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$ and together with the filter condition to $\mu \cup \mu' \in [\![\text{FILTER}(\text{AND}(P_1, P_2), R)]\!]_G$. By induction hypothesis and Assumption 1, this means that $G \cup \mathcal{R} \models_{n_3} (m(P_1) \cup m(P_2) \cup \overline{ft}(R))(\mu \cup \mu')$ and especially, $(m(P_2) \cup \overline{ft}(R))(\mu \cup \mu') \in close(G \cup \mathcal{R})$. That is, μ' is a mapping such that $close(G \cup \mathcal{R}) \models_b ((m(P_2) \cup \overline{ft}(R))\mu)\mu'$. This contradicts $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, m(P_2) \cup \overline{ft}(R)\rangle\mu$.

(6) *Let P be of the form* $FE(P_1, P_2)$*:*

"⇒" Let $\mu \in [\![\text{FE}(P_1, P_2)]\!]_G$, i.e., $\mu \in [\![P_1]\!]_G$ and there exists μ' such that $\mu' \in [\![P_2\mu]\!]_G$. By the induction hypothesis, that means that μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$ and $G \cup \mathcal{R} \models_{n_3} (m(P_2)\mu)\mu'$. As the relabeling function rl only changes the name of variables outside the domain of μ, we can also find a mapping μ'' such that $G \cup \mathcal{R} \models_{n_3} (rl(m(P_2, sv(P_1)))\mu)\mu''$. As the arguments of the predicate incl are scoped locally, this means that $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{incl}, rl(m(P_2), sv(P_1))\rangle\mu$ and thus $G \cup \mathcal{R} \models_{n_3} m(P)\mu$.

"⇐" Let μ be a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P)\mu$. Then, $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$ and μ is a minimal mapping fulfilling this condition. To prove the latter, assume that there is a mapping $\mu' \subsetneq \mu$ such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu'$. But then $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\mu'$, which contradicts the minimality of μ. We thus know by induction hypothesis that $\mu \in [\![P_1]\!]_G$. As the $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\mu$, we furthermore know that if $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$, there exists a mapping A such that $\mathfrak{I}[A] \models_{n_3} rl(m(P_2), sv(P_1))\rangle\rangle\mu$ which implies $\mathfrak{I} \models_{n_3} (rl(m(P_2), sv(P_1))\rangle\rangle\mu)A_2$ if we treat A_2 as a substitution. Let σ be the renaming function such that $m(P_2)\sigma = rl(m(P_2), sv(P_1))$. Then we have for $\mu' = A_2 \circ \sigma^{-1}$ that $\mathfrak{I} \models_{n_3} m(P_2)\mu\mu'$. We can now find a minimal mapping $\mu'' \subseteq \mu'$ such that $G \cup \mathcal{R} \models_{n_3} (m(P_2)\mu)\mu''$ and get by induction hypothesis that $\mu'' \in [\![m(P_2)\mu]\!]_G$.

(7) *Let P be of the form* FNE(P_1, P_2)*:*

"⇒" Let $\mu \in [\![P]\!]_G$. That means that $\mu \in [\![P_1]\!]_G$ and $[\![P_2\mu]\!]_G = \emptyset$. By induction hypothesis μ is a minimal mapping such that $G \cup \mathcal{R} \models_{n_3} m(P_1)\mu$. To show that $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\mu$ we assume the contrary, namely that $close(G \cup \mathcal{R}) \models_b rl(m(P_2), sv(P_1))\mu$. That means that for all interpretations such that $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$ we can find a mapping A such that $\mathfrak{I}[A] \models_{n_3} rl(m(P_2), sv(P_1))\mu$. With the injective mapping σ such that $m(P_2)\sigma = rl(m(P_2), sv(P_1))$, we again get that $G \cup \mathcal{R} \models_{n_3} (m(P_2)\mu)\sigma A_2$ it thus exists a minimal mapping μ' such that $G \cup \mathcal{R} \models_{n_3} (m(P_2)\mu)\mu'$ which leads, by induction hypothesis, to $\mu' \in [\![P_2\mu]\!]_G$. This contradicts $[\![P_2]\!]_G = \emptyset$.

"⇐" Let μ be minimal such that $G \cup \mathcal{R} \models_{n_3} m(P)$. Then $G \cup \mathcal{R} \models_{n_3} m(P_1)$. Since all variables which do not occur in P_1 are scoped locally wice know that μ is minimal and can thus conclude using the induction hypothesis that $\mu \in [\![P_1]\!]_G$. Let us assume that there exists a minimal mapping $\mu' \in [\![P_2\mu]\!]_G$. Then we know, again by induction hypothesis, that $G \cup \mathcal{R} \models_{n_3} m(P_2)\mu\mu'$. But then, it holds that $close(G \cup \mathcal{R}) \models_b m(P_2)\mu\mu'$ and with mapping σ such that $m(P_2)\sigma = rl(m(P_2), sv(P_1))$ we get $close(G \cup \mathcal{R}) \models_b (rl(m(P_2), sv(P_1))\mu)\sigma^{-1}\mu'$. This contradicts $G \cup \mathcal{R} \models_{n_3} \langle \_, \text{notIncl}, \langle rl(m(P_2), sv(P_1))\rangle\rangle\mu$.

(8) *For P be of the form* FILTER(P, R) the claim is equivalent to Assumption 1.

6.2 From queries to consequences

Given a SPARQL query "$(n\ x)$ WHERE P", the conclusion mapping h takes the query form $f = (n\ x)$ and the SPARQL graph pattern P as input, and produces an N3 graph pattern as output. Let F be the set of query forms and "list" a function which, when provided with a set of variables V, produces an ordered list of these. We define the conclusion mapping $h : F \times GP \to N3P$ as follows:

$$\begin{aligned} h((\text{CONSTRUCT}, Q), P) &= Q, \\ h((\text{SELECT}, l), P) &= \{\langle \_, \text{result}, l\rangle\}, \quad l \neq *, \\ h((\text{SELECT}, *), P) &= \{\langle \_, \text{result}, \text{list}(sv(P))\rangle\}. \end{aligned}$$

where _ stands for an arbitrary blank node. Note that we use the scoping function sv from above to resolve the asterisk.

For our example query in Listing 9, our mapping produces the consequence[20]

$$\begin{aligned} h((SELECT, *), P) = \{\langle \_, \text{result}, \text{list}(sv(P))\rangle = \\ \{\langle \_, \text{result}, (?x\ ?y)\rangle\}. \end{aligned}$$

When also considering the results from the premise mapping m, the fully translated query $m(P) \to h((n\ x), P)$ gives us the following N3 rule:

$$\begin{aligned} &\{\langle ?x, p1, ?y\rangle, \\ &\quad \langle \_ \text{notIncl}, \{\langle ?x, p2, ?z'\rangle, \\ &\qquad\qquad \langle \_, \text{notIncl}, \{\langle ?x, p3, ?y'\rangle\}\rangle\}\rangle\} \\ &\to \{\langle \_, \text{result}, (?x\ ?y)\rangle\} \end{aligned}$$

We can use Lemma 3 to prove the following:

Lemma 4. *Given the runtime rules $\mathcal{R}$ for the predicate "*union*", a proper RDF graph G, a query* (CONSTRUCT Q) WHERE P *where P is well designed and does not contain property path expressions, and a mapping $\mu : X \to I \cup L$. If $\mu \in [\![P]\!]_G$, then*

$$G \cup \mathcal{R} \cup \{m(P) \to h((\text{CONSTRUCT } Q), P)\} \models_{n_3} Q\mu$$

Proof. We first note that $h((\text{CONSTRUCT } Q), P) = Q$. Let $\mathfrak{I} \models_{n_3} G \cup \mathcal{R} \cup \{m(P) \to Q\}$. From $\mu \in [\![P]\!]_G$ follows by Lemma 3 that $\mathfrak{I} \models_{n_3} m(P)\mu$. As $\mathfrak{I} \models m(P) \to Q$ and the variables in the rule are universally quantified, we can apply the rule using a mapping A^μ depending on μ. We define $A^\mu(x) := (\mathfrak{I}(\mu'(x)), \mu'(x))$. If $Q\mu$ contains blank nodes, then these are quantified locally and they do not depend on the premise.

With the lemma we can apply a single translated N3 rule to a ground RDF graph and obtain the same triples the original query SPARQL provides. When applying the rules recursively, we rely on N3 semantics. Note that the result can easily be extended to RDF graphs containing blank nodes as N3 rules never change the scoping of blank nodes in plain RDF graphs and these are quantified on graph level.

6.3 SPARQL Concepts in N3

Since some SPARQL features do not have an equivalent N3 construct, we manually implement those in a "runtime" ruleset so as to still allow for the one query–one rule principle. We describe these rules below.

Union construct. For UNION, a single translated N3 rule will include a triple `<a> union <b>` (Section 6.1, *Mapping m from SPARQL to N3 Graph Patterns*). This triple will be resolved by the following runtime rules:[21][22]

$$\begin{aligned} \langle ?x, \text{union}, ?y\rangle &\leftarrow ?x \\ \langle ?x, \text{union}, ?y\rangle &\leftarrow ?y \end{aligned}$$

[20] In our implementation, the result triple also includes the original SELECT variable names as strings, so query results are returned in a way similar to SPARQL.

[21] Note that N3 allows forward and backward chaining rules; we write them as backward rules here.

[22] In the implementation the body variable is called by an extra-logical predicate as not all reasoners support variables as rule premises. We omit this operational detail here.

To back up Lemmas 3 we now prove **Assumption 2**.

Lemma 5. Assumption 2. *Given an RDF ground graph G, the set of run time rules for "*union*" $\mathcal{R}$ from above and two N3 graph patterns P_1 and P_2, then:*

$$G \cup \mathcal{R} \models_{n_3} \langle\langle P_1\rangle, \text{union}, \langle P_2\rangle\rangle$$
$$\textit{iff}$$
$$G \cup \mathcal{R} \models_{n_3} P_1 \textit{ or } G \cup \mathcal{R} \models_{n_3} P_2$$

Proof. "⇒" Let $G \cup \mathcal{R} \models_{n_3} \langle\langle P_1\rangle, \text{union}, \langle P_2\rangle\rangle$. We construct the Herbrand Model of $G \cup \mathcal{R}$, that is, we set $\Delta_{\mathfrak{I}} = T_c$ and $\mathfrak{I}(t) = t$ for $t \in U \cup L$ for all other terms $\mathfrak{I}$ behaves as specified in Section 5.2. $IEXT$ consists of G and all triples we can retrieve by applying the rules in $\mathcal{R}$ multiple times. These triples are of the form $\langle\langle P_1\rangle, \text{union}, \langle P_2\rangle\rangle$ where $G \cup \mathcal{R} \models_{n_3} P_1$ or $G \cup \mathcal{R} \models_{n_3} P_2$.
"⇐" Let $\mathfrak{I} \models_{n_3} G \cup \mathcal{R}$, the $\mathfrak{I} \models_{n_3}$ the premise or one of the two rules in $\mathcal{R}$ and we thus get $\mathfrak{I} \models_{n_3} \langle\langle P_1\rangle, \text{union}, \langle P_2\rangle\rangle$.

Note that even if we apply the rules forwardly of $\mathcal{R}$ on only one single triple $\langle s, p, o\rangle$, the application produces an infinite number of consequences

$$\langle\langle s,p,o\rangle, \text{union}, \langle s,p,o\rangle\rangle,$$
$$\langle\langle s,p,o\rangle, \text{union}, \langle s,p,o\rangle\rangle, \text{union}, \langle s,p,o\rangle\rangle,$$
$$\langle\langle s,p,o\rangle, \text{union}, \langle s,p,o\rangle, \text{union}, \langle s,p,o\rangle\rangle, \text{union}, \langle s,p,o\rangle\rangle,$$
$$\ldots$$

We, therefore, evaluate these rules using backward chaining in our implementation.

Recursive property paths. Recursive property paths are translated by the pp_{N3} function to an N3 list predicate (Section 6.1). These list predicates are resolved by the runtime rules in Figure 1. The rules use two list builtins, namely `list:notmember` ("notmember") and `list:append` ("append"). These builtins are described in the online specification (5). We further make two important notes:

Variable predicates. Rules (2)-(16) use the *isvar* predicate to ensure that the property path variable (e.g., $?p$) has been bound. To illustrate the potential problem, consider an example rule with a variable predicate $?r$: $\langle jane,\ ?r,\ alice\rangle \rightarrow \langle ?r,\ \text{type},\ \text{relation}\rangle$. To resolve the body triple, for instance, rule (16) could be called, which binds $?r$ to $(?p$ "+"). This means that the rule's property path variable $?p$ is not actually bound. Resolving the rule's second body triple may involve calling rule (17); to resolve that rule's body triple, of which $?p$ is not bound, rule (16) may again be called, leading to infinite solutions.

Cyclic rule applications. Recursive property paths could easily lead to nontermination due to cyclic rule application.

$\langle ?s,\ (?p1\ \text{"/"}\ ?p2),\ ?o\rangle$ (2)
$\leftarrow \langle ?p1,\ \text{isvar},\ \text{false}\rangle, \langle ?p2,\ \text{isvar},\ \text{false}\rangle,$
$\langle ?s,\ ?p1,\ ?o1\rangle, \langle ?o1, ?p2, ?o\rangle$
$\langle ?s,\ (?p1\ \text{"—"}\ ?p2),\ ?o\rangle \leftarrow \langle ?p1,\ \text{isvar},\ \text{false}\rangle,\ \langle ?s, ?p1, ?o\rangle.$ (3)
$\langle ?s,\ (?p1\ \text{"—"}\ ?p2),\ ?o\rangle \leftarrow \langle ?p2,\ \text{isvar},\ \text{false}\rangle,\ \langle ?s, ?p2, ?o\rangle.$ (4)
$\langle ?s,\ (\text{"^"}?p),\ ?o\rangle \leftarrow \langle ?o,\ \text{isvar},\ \text{false}\rangle, \langle ?o,\ ?p,\ ?s\rangle$ (5)
$\langle ?s,\ (\text{"!"}?p),\ ?o\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle,$ (6)
$\langle ?s,\ ?p1,\ ?o\rangle, \langle ?p,\ \text{noteq},\ ?p1\rangle$
$\langle ?s,\ (\text{"!"}(\text{"^"}?p)),\ ?o\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle,$ (7)
$\langle ?o,\ ?p1,\ ?s\rangle, \langle ?p,\ \text{noteq},\ ?p1\rangle$
$\langle ?s,\ (\text{"!"}?l),\ ?o\rangle \leftarrow \langle ?l,\ \text{rawtype},\ \text{list}\rangle,$ (8)
$\langle ?s,\ ?p,\ ?o\rangle, \langle ?l,\ \text{notmember},\ ?p\rangle$
$\langle ?s,\ (\text{"!"}\ ?l),\ ?o\rangle \leftarrow \langle ?l,\ \text{rawtype},\ \text{list}\rangle,$ (9)
$\langle ?o,\ ?p,\ ?s\rangle, \langle ?l,\ \text{notmember},\ (\text{"^"}?p)\rangle$
$\langle ?s,\ (?p\ \text{"?"}),\ ?s\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?s, ?p, ?x\rangle$ (10)
$\langle ?s,\ (?p\ \text{"?"}),\ ?s\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?x, ?p, ?s\rangle$ (11)
$\langle ?s,\ (?p\ \text{"?"}),\ ?o\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?s,\ ?p,\ ?o\rangle$ (12)
$\langle ?s,\ (?p\ \text{"*"}),\ ?s\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?s, ?p, ?x\rangle$ (13)
$\langle ?s,\ (?p\ \text{"*"}),\ ?s\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?x, ?p, ?s\rangle$ (14)
$\langle ?s,\ (?p\ \text{"*"}),\ ?o\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle, \langle ?s,\ (?p\ \text{"+"}),\ ?o\rangle$ (15)
$\langle ?s,\ (?p\ \text{"+"}),\ ?o\rangle \leftarrow \langle ?p,\ \text{isvar},\ \text{false}\rangle,$ (16)
$\langle ?s,\ (?p\ \text{"+"}\ ()),\ ?o\rangle$
$\langle ?s,\ (?p\ \text{"+"}\ ?l),\ ?o\rangle \leftarrow \langle ?s,\ ?p,\ ?o\rangle$ (17)
$\langle ?s,\ (?p\ \text{"+"}\ ?l),\ ?o\rangle \leftarrow \langle ?s,\ ?p,\ ?o1\rangle,$ (18)
$\langle ?l,\ \text{notmember},\ ?o1\rangle,$
$\langle (?l\ (?o1)),\ \text{append},\ ?l2\rangle,$
$\langle ?o1,\ (?p\ \text{"+"}\ ?l2),\ ?o\rangle$

Figure 1. Runtime rules for recursive property paths.

To illustrate this, consider the triples $\langle$alice, sibling, jane$\rangle$ and $\langle$jane, sibling, alice$\rangle$, and recursive property path $\langle ?s,\ (\text{sibling "+"}),\ ?o\rangle$. Without cycle detection, when applying rule (18), variables $?s, ?o1$ could be respectively bound to jane, alice (first triple); in the first recursive call, variable $?s, ?o1$ are bound to alice, jane (second triple); in the second recursive call, variable $?s, ?o1$ would again be bound to jane, alice (first triple). We thus see the start of a cycle that will not terminate. To avoid this, we add cycle detection to rule (18): list $?l$ keeps track of visited objects $?o1$ and the rule stops when a node is revisited.

7 Evaluation

We implemented SPARQL-in-N3 (SiN3) and evaluated its performance by comparing it with systems from the state of the art. All code and artefacts are available on github (*11*), including a demo of the Zika use case (see below).

7.1 Evaluation setup

Evaluated systems. We evaluate SiN3, labelled *sin3*, using the eye v10.24.10 reasoner (*14*), which can perform forward and backward reasoning. Our prior demo paper (*9*) outlines the implementation of *sin3*, including the execution steps and rulesets involved. We note that each experiment was executed 5 times and results were averaged.

Using CONSTRUCT *queries for rule-based reasoning*. We compare *sin3* with two other systems that utilize CONSTRUCT queries for rule-based reasoning, namely *spinrdf* (*10*) and *recSPARQL* (*17*). The work by Polleres (*15*) was implemented using an older version of dlvhex which we found to no longer be compatible with modern OS. As far as we know, the work by Gottlob et al. (*8*) was not implemented; and SparqLog (*7*) does not support recursive querying.

Property path implementation. We separately compare the *sin3* implementation of property paths with Apache Jena (*jena*) (*27*). We note that the *spinrdf* and *recSPARQL* systems are built on top of *jena*; meaning the two aforementioned systems would actually rely on *jena* for executing property paths within SPARQL queries.

Hardware. The experiments were conducted on a MacBook Pro with an Apple M1 Pro processor, 32 GB of RAM, and a 1 TB SSD, running macOS Sonoma 14.6.1.

Compliance. Regarding compliance with the SPARQL 1.1 specification (*6*), we currently lack support for the GROUP BY and HAVING clauses, named graphs, the BIND clause, expressions (incl. aggregate functions) in SELECT clauses, and LIMIT clauses in subqueries.

Datasets and queries Our evaluation covers the use cases listed below.

LMDB and YAGO. We re-use the setup from recSPARQL (*17*), which involved the *LMDB* (Linked Movie Database), an RDF graph describing movies and actors (*37*) (6,148,121 triples); and *YAGO* (Yet Another Great Ontology), an RDF graph describing people, locations, and movies (*38*) (3,000,006 triples). The authors formulated recursive CONSTRUCT queries for both datasets:

Bac1. Return actors with a "Bacon number", meaning they either appeared in a movie together with Kevin Bacon, or appeared with another actor with a "Bacon number".

Bac2. Same as *Bac1* but searches for co-appearances in movies with the same director.

Bac3. Same as *Bac2*, but requires those directors to also be actors (not necessarily in the same movie).

They also formulated CONSTRUCT queries for YAGO alone:

Geo. Return places in which the city of Berlin is (directly and indirectly) located.

MarrUs. Return people who are (directly and indirectly) related, through the *isMarriedTo* relation, to someone who owns property in the United States.

The above queries thus mostly calculate the transitive closure of a given relation.

Zika screening. In a previous healthcare use case (*9*), we used CONSTRUCT queries to implement the CDC testing guidance for Zika. Here, we expand this test case with randomly generated datasets, different versions of the HL7 FHIR vocabulary (*39*), and reasoning over a biomedical ontology, i.e., SNOMED (*40*). This use case illustrates *modularity*, with separate queries each checking for a Zika indicator (their results being re-used in some cases) and "utility" queries that facilitate the navigation of FHIR; and *expressivity*, by incorporating a custom entailment regime (OWL2 RL (*12*)).

Property paths. We used the gMark (*41*) graph instance and workload generator to generate a small-scale synthetic dataset and synthetic queries using property paths. In particular, we generated a graph of 40 nodes with fixed typing (researchers, papers, journals, conferences, and cities) and four predicates (authors, publishedIn, heldIn, and extendedTo), while using the schema to restrict admissible type pairs and enforce fixed in/out degrees (e.g., exactly 3 *authors* per paper and exactly 1 *publishedIn* / *extendedTo* edge per paper). Furthermore, to serve as non-synthetic examples of property paths, we isolated and re-used multiple versions of the property paths used in the Zika screening use case.

7.2 Results

We report wall-clock runtimes averaged over five runs and, where applicable, decompose them into data loading, query compilation, and execution (reasoning). The results below summarize the main performance trends across (i) recursive transitive-closure benchmarks (LMDB/YAGO), (ii) the Zika screening workload (including OWL2 RL reasoning), and (iii) property paths.

LMDB and YAGO Table 3 shows the evaluation results for LMDB and YAGO. *Sin3* and *spinrdf* start by loading the dataset into memory (Load), while *recSPARQL* pre-creates a persistent Jena Triple DataBase (Load TDB). *Sin3* and *spinrdf* then convert input SPARQL queries into SPIN code (Gen SPIN); next, *sin3* converts the SPIN code into N3 rules (Gen N3).

Sin3 takes much longer for Gen SPIN than *spinrdf*, as *sin3* requires starting a separate JVM for generating SPIN code. *Sin3* is competitive regarding reasoning performance (Exec): for the LMDB dataset, on average, *sin3* takes ca. 10.5s, *spinrdf* ca. 11.8s, and recSPARQL ca. 36.2s. For the Yago dataset, on average, *sin3* takes ca. 6.7s, *spinrdf* ca. 3.1s, and recSPARQL ca. 34.7s. We thus observe that *spinrdf*, which relies on a query engine (Jena), performs much better for the Yago dataset. This is revisited in our conclusion.

LMDB									
Query	**sin3**				**spinrdf**			**recSPARQL**	
	Load	Gen SPIN	Gen N3	Exec	Load	Gen SPIN	Exec	Load (TDB)	Exec
Bac1	31	0.58	0.10	20.1	26.9	0.06	12	75.4	65.7
Bac2		0.58	0.08	2.4		0.02	3.9		18.4
Bac3		0.58	0.08	9.0		0.02	19.5		24.5
YAGO									
Bac1	36.1	0.58	0.08	14.3	44.9	0.02	7.5	113686	50.2
Bac2		0.57	0.08	1.5		0.003	0.01		3.2
Bac3		0.57	0.08	5.0		0.003	2.8		22.4
Geo		0.57	0.07	0.4		0.003	0.004		14.4
MarrUs		0.57	0.08	12.3		0.003	5.1		83.2

Table 3. Execution times for LMDB and YAGO queries (times in seconds).

System	$data_{0.1}^{orig}$	$data_{0.2}^{orig}$	$data_{0.1}^{red}$	$data_{0.2}^{red}$	$data_{0.2}^{red,subcl}$
sin3-bwd	9.4	229	0.01	0.4	0.8
sin3-fwd	13.2	32.2	0.6	1.8	3.8
spinrdf	0.5	1.4	0.3	0.8	5.9

Table 4. Execution times for Zika queries (times in seconds).

Zika Screening Table 4 shows the evaluation results for the Zika screening use case. We differentiate the results based on the operational type of reasoning (forward vs. backward) and implementation (rule engine vs. query engine). We refer to SiN3 using backward reasoning as *sin3-bwd*, forward reasoning as *sin3-fwd*, and use *spinrdf* as the exemplar recursive query engine, as it performed best in the prior experiment. We further used the following FHIR vocabularies:

Original FHIR vocabulary. The original FHIR vocabulary is deeply nested, meaning any non-trivial query will require a lot of joins. We thus expect query engines (*spinrdf*) to have an advantage as they tend to be optimized for joins.

We generated 2 random datasets of 1000 patients, who respectively had 0.1 chance ($data_{0.1}^{orig}$) and 0.2 chance ($data_{0.2}^{orig}$) of having a Zika indicator. This led to respectively ca. 0.5% of patients ($data_{0.1}^{orig}$) and ca. 2.5% ($data_{0.2}^{orig}$) to be tested for Zika. Patients have at most 2 conditions.

Indeed, we found the number of joins to be problematic for rule engines. Reasoning over $data_{0.1}^{orig}$ using *sin3* with forward and backward reasoning respectively takes ca. 13.2s and 9.4s. *Spinrdf* only takes ca. 0.5s. For $data_{0.1}^{orig}$, *sin3-bwd* has an advantage over the *sin3-fwd* variant, as only a small number of patients will be positive, and thus only a small part of the state space has to be searched. This advantage is totally lost for $data_{0.2}^{orig}$, where *sin3-bwd* takes ca. 229s, *sin3-fwd* takes ca. 32s, and *spinrdf* only takes ca. 1.6s.

Reduced FHIR vocabulary. A reduced version of the FHIR vocabulary relied less on deeply structured data and is described online (*11*). We similarly generated random datasets using this vocabulary, namely $data_{0.1}^{red}$ and $data_{0.2}^{red}$.

Here, we see the performance equalizing. For $data_{0.2}^{red}$, which triggers the most rules, *sin3-bwd* takes ca. 0.4s, *sin3-fwd* takes ca. 1.8s, and *spinrdf* takes ca. 0.8s. We thus observe that complex and deeply nested data structures, with non-trivial data size, are problematic for rule engines. We revisit this in future work.

Reasoning with SNOMED and OWL2 RL. The CDC guidelines, on which this use case is based, refer to high-level condition codes from SNOMED (*40*). To accommodate this, patients in the above generated datasets were also tagged with high-level condition codes (e.g., muscle pain). In practice, however, patients will more likely be tagged with a specific sub-condition (e.g., abdominal muscle pain). In this experiment, we randomly generated a third dataset $data_{0.2}^{red,subcl}$, which includes more specific condition codes (and also uses the reduced FHIR vocabulary). We use ontology-based reasoning to close the gap between the CDC screening rules, which refer to high-level codes, and the specific condition codes in the dataset. To that end, we extracted the SNOMED subclass hierarchy (31.5 Mb) and included it in the dataset; and added a CONSTRUCT query for the OWL2 RL *cax-sco* OWL2 RL rule (*12*) to the ruleset.

We expect a backward reasoner to perform better here, as it will not have to materialize the entire subclass closure. Indeed, *sin3-bwd* takes around 0.8s, *sin3-fwd* around 3.8s, and *spinrdf* around 5.9s. To further confirm this observation, we conducted an extra experiment with the Deep Taxonomy[23], which requires a much larger materialization of the subclass hierarchy. Here, the differences are much more

[23] https://eulersharp.sourceforge.net/2009/12dtb/

pronounced: *eye-bwd* takes avg. 34ms to answer a query for all instances of a particular type, whereas *spinrdf* does not return results after 1h. Hence, we observe that large transitive closures can be problematic for forward reasoners, such as *spinrdf*.

	gMark		**zika**	
	avg	*stdev*	*avg*	*stdev*
sin3*	1.23	6.0	0.98	1.3
jena	0.32	0.47	0.5	0.02

Table 5. Execution times for property paths (time in seconds). ($*$ leaving out 10 gMark queries that did not finish after 5 minutes)

Property paths Table 5 shows initial evaluation results for property paths. *Jena* performs better for property paths overall, taking only ca. 0.32s (*gMark*) and 0.5s (*zika*) vs. ca. 1.23s (*gMark*) and 0.98s (*zika*) for *sin3*. Moreover, for *sin3*, 10 queries of the *gMark* case did not finish after 5 minutes. *Sin3* also shows much more variability in performance, as illustrated by the standard deviation (stdev) values; up to ca. 6s for *zika*.

The 10 aforementioned *gMark* queries included property paths with recursion (i.e., $*, +$), which are implemented via our runtime (Section 6.3, *Recursive property paths*). We revisit these performance issues in future work. At the same time, we do point out that the *gMark* queries do not reflect typical usage scenarios; they include highly artificial property paths that "oscillate" multiple times between the same nodes, i.e., using regular and reverse paths.

8 Conclusions

To facilitate rule-based reasoning on the SW, the familiarity and popularity of the SPARQL query language can be leveraged to express logical rules. This paper presented *SiN3*, which implements rule-based reasoning with SPARQL CONSTRUCT queries by translating them into the Notation3 (N3) rule language. The translated rules can be reasoned over using existing N3 machinery, using both bottom-up and top-down execution. By allowing queries to act on their own and other queries' results, this translation also adds full recursion to SPARQL.

To the best of our knowledge, this is the first work to propose a translation of SPARQL to a SW rule language that follows the RDF data model. Compared to Datalog translations, SPARQL queries are translated into N3 rules that directly operate on triples. Firstly, this means there is no need to translate a triple-based RDF dataset into Datalog facts before reasoning. In fact, the SiN3 translation starts from a triple-based encoding of SPARQL queries (SPIN) (9); N3 rules inspect these SPIN queries and generate corresponding N3 rules via the N3 meta-reasoning capabilities. Hence, aside from the initial SPARQL parsing, the SiN3 translation is entirely triple-based, and both the translation and execution take place within N3. Secondly, since the produced N3 rules rely on SW technology, translated rules are exchangeable and interoperable. In this vein, we further target a one query–one rule principle. Instead of generating additional rules to implement missing SPARQL constructs, we present the novel concept of a runtime ruleset.

Compared to reasoning extensions of SPARQL engines, which focus on forward (bottom-up) reasoning, a rule language translation allows re-using its existing forward and/or backward (top-down) reasoning machinery. Our evaluation shows the competitive performance of SiN3 for recursive SPARQL queries compared to such query engine extensions. Notably, the evaluation also distinguishes the benefits of query engines versus backward rule engines to implement rule-based reasoning. We observe that backward reasoning can be essential when dealing with large search spaces, but query engines are much better at dealing with large datasets, many joins, and property paths.

We aim for our work to be an initial step in the cross-pollination of queries and rules on the SW. Observations relevant to this aim are found across our work. The Zika use case highlights the benefits of extending SPARQL queries with rule-based reasoning; it adds recursivity, improves modularity by encapsulating reusable logic within distinct rules, and increases expressivity by allowing custom entailment regimes. Our evaluation, as mentioned above, compares the benefits of query engines with those of rule-based reasoners across multiple use cases. When discussing our translation, we also highlight the semantic mismatch between query languages, which are centred on variable mappings, and rule languages, which deal with instantiated patterns. Insights from our empirical experiments and rule translation could lead to new work that reduces the gap between SW querying and rule-based reasoning.

Many avenues for future work remain, several of which are already underway. Our evaluation showed that SiN3 performs poorly on recursive property paths: the generality of the runtime rules can be costly, requiring many variable instantiations and, for nested property paths, repeated rule calls. A promising alternative is to inspect the query's property paths and use N3 meta-reasoning to pre-compute a more concrete, query-specific ruleset, and then reason over these pre-computed rules. We already implemented an initial version of this approach and observed speedups of orders of magnitude compared to the current runtime rules.

We aim to lift the current restrictions on MINUS and OPTIONAL. For OPTIONAL in particular, we are developing runtime rules that support an unrestricted version. We also plan to finish our implementation of remaining SPARQL features, including aggregation (aggregate functions in SELECT, and GROUP BY and HAVING) and BIND, among others. To further encourage cross-pollination between query

and rule-based features, we intend to extend the runtime with additional SPARQL constructs so they can be used in rules.

We also plan to clarify and extend our treatment of SPARQL multiset (bag) semantics. The current SiN3 translation and experiments effectively assume set semantics: we do not track solution multiplicities, and derived knowledge is handled as RDF graphs. While adequate for the reasoning tasks considered here, bag-sensitive behavior remains out of scope. Future work will investigate multiplicity-aware encodings, for example, by annotating derived triples with counts or provenance to preserve multiplicities when needed, and enable faithful bag-based aggregation.

Finally, we do not cover the corner case of recursion combined with negation, namely recursive derivations that depend on negation-as-failure. Extending SiN3's formalization and runtime to support and empirically assess such recursion-and-negation interactions, for example via stratified or well-founded treatments, is left for future work.

References

1. Motik, B., P. Patel-Schneider, and B. Cuenca Grau. *OWL 2 Web Ontology Language Direct Semantics (Second Edition)*. W3C recommendation, W3C, 2012. URL https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.
2. Horrocks, I., P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C member submission, W3C, 2004. URL https://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.
3. Lanthaler, M., D. Wood, and R. Cyganiak. *RDF 1.1 Concepts and Abstract Syntax*. W3C recommendation, W3C, 2014. URL https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.
4. Kifer, M. Rule Interchange Format: The Framework. In *Web Reasoning and Rule Systems* (D. Calvanese and G. Lausen, eds.). Springer, Berlin, Heidelberg, 2008, pp. 1–11. doi:10.1007/978-3-540-88808-6_1.
5. Van Woensel, W., D. Arndt, P.-A. Champin, D. Tomaszuk, and G. Kellogg. *Notation3 Language*. W3C community group report, W3C, 2023. URL https://w3c.github.io/N3/reports/20230703/.
6. Harris, S. and A. Seaborne. *SPARQL 1.1 Query Language*. W3C recommendation, W3C, 2013. URL https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.
7. Angles, R., G. Gottlob, A. Pavlović, R. Pichler, and E. Sallinger. SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog. *Proc. VLDB Endow.*, Vol. 16, No. 13, 2023, pp. 4240–4253. doi:10.14778/3625054.3625061.
8. Gottlob, G. and A. Pieris. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*. AAAI Press, 2015, pp. 2999–3007. doi:10.5555/2832581.2832668. URL https://www.ijcai.org/Proceedings/15/Papers/424.pdf.
9. Arndt, D., W. Van Woensel, and D. Tomaszuk. SiN3: Scalable Inferencing with SPARQL CONSTRUCT Queries. In *Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks*, *CEUR Workshop Proceedings*, Vol. 3632. CEUR-WS.org, 2023. URL https://ceur-ws.org/Vol-3632/ISWC2023_paper_469.pdf.
10. Seaborne, A. and M. Jusevičius. SpinRDF, 2019. URL https://github.com/spinrdf/spinrdf.
11. Arndt, D., W. Van Woensel, and D. Tomaszuk. SiN3 (SPARQL in N3) GitHub Repository. URL https://github.com/william-vw/SiN3.
12. Calvanese, D., J. Carroll, G. De Giacomo, J. Hendler, I. Herman, B. Parsia, P. F. Patel-Schneider, A. Ruttenberg, U. Sattler, and M. Schneider. *OWL 2 Web Ontology Language Profiles (Second Edition): OWL 2 RL*. W3C recommendation, W3C, 2012. URL https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.
13. Arndt, D., W. Van Woensel, and D. Tomaszuk. SPARQL in N3: SPARQL CONSTRUCT as a Rule Language for the Semantic Web. In *Rules and Reasoning: 9th International Joint Conference, RuleML+RR 2025, Istanbul, Turkey, September 22–24, 2025, Proceedings*, *Lecture Notes in Computer Science*, Vol. 16144. Springer, 2026, pp. 209–226. doi:10.1007/978-3-032-08887-1_13.
14. Verborgh, R. and J. De Roo. Drawing Conclusions from Linked Data on the Web: The EYE Reasoner. *IEEE Software*, Vol. 32, No. 3, 2015, pp. 23–27. doi:10.1109/MS.2015.63.
15. Polleres, A. From SPARQL to Rules (and Back). In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*. ACM, 2007, pp. 787–796. doi:10.1145/1242572.1242679.
16. Kostylev, E. V., J. L. Reutter, and M. Ugarte. CONSTRUCT Queries in SPARQL. In *18th International Conference on Database Theory (ICDT 2015)*, *Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 212–229. doi:10.4230/LIPIcs.ICDT.2015.212.
17. Reutter, J., A. Soto, and D. Vrgoč. Recursion in SPARQL. *Semantic Web*, Vol. 12, No. 5, 2021, pp. 711–740. doi:10.3233/SW-200401.
18. Knublauch, H., J. A. Hendler, and K. Idehen. SPIN RDF, 2011. URL https://www.w3.org/submissions/2011/SUBM-spin-overview-20110222/. W3C Submission.
19. Hogan, A., J. Reutter, and A. Soto. Recursive SPARQL for Graph Analytics. *arXiv preprint arXiv:2004.01816*. URL https://arxiv.org/abs/2004.01816.
20. Corby, O., C. Faron-Zucker, and F. Gandon. LDScript: A Linked Data Script Language. In *International Semantic Web Conference (ISWC 2017)*. Springer, 2017, pp. 208–224. doi:10.1007/978-3-319-68288-4_13.

21. Atzori, M. Computing Recursive SPARQL Queries. In *2014 IEEE International Conference on Semantic Computing (ICSC)*. IEEE, 2014, pp. 258–259. doi:10.1109/ICSC.2014.54.
22. Pérez, J., M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. *Journal of Web Semantics*, Vol. 8, No. 4, 2010, pp. 255–270. doi:10.1016/j.websem.2010.01.002.
23. Kochut, K. J. and M. Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *The Semantic Web: 4th European Semantic Web Conference (ESWC 2007)*. Springer, 2007, pp. 145–159. doi:10.1007/978-3-540-72667-8_12.
24. Libkin, L., J. Reutter, and D. Vrgoč. Trial for RDF: Adapting Graph Query Languages for RDF Data. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2013)*. ACM, 2013, pp. 201–212. doi:10.1145/2463664.2465226.
25. Arenas, M., G. Gottlob, and A. Pieris. Expressive Languages for Querying the Semantic Web. *ACM Trans. Database Syst.*, Vol. 43, No. 3. doi:10.1145/3238304.
26. Bellomarini, L., E. Sallinger, and G. Gottlob. The Vadalog System: Datalog-Based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.*, Vol. 11, No. 9, 2018, pp. 975–987. doi:10.14778/3213880.3213888.
27. Apache. Apache Jena, 2021. URL https://jena.apache.org/.
28. Cérès, R., O. Corby, and F. Gandon. Corese, 2023. URL https://github.com/Wimmics/corese.
29. Robie, J., M. Dyck, and J. Spiegel. XML Path Language (XPath) 3.1, 2017. URL https://www.w3.org/TR/xpath-31/. W3C Recommendation.
30. Pérez, J., M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, Vol. 34, No. 3, 2009, pp. 1–45. doi:10.1145/1567274.1567278.
31. Polleres, A. and J. P. Wallner. On the Relation Between SPARQL 1.1 and Answer Set Programming. *Journal of Applied Non-Classical Logics*, Vol. 23, No. 1-2, 2013, pp. 159–212. doi:10.1080/11663081.2013.798992.
32. Salas, J. and A. Hogan. Semantics and Canonicalisation of SPARQL 1.1. *Semantic Web*, Vol. 13, No. 5, 2022, pp. 829–893. doi:10.3233/SW-212871.
33. The SPARQL Exists CG. *SPARQL Exists Report*. W3C community group report, W3C, 2019. URL https://w3c.github.io/sparql-exists/docs/sparql-exists.html.
34. Prud'hommeaux, E. and G. Carothers. *RDF 1.1 Turtle*. W3C recommendation, W3C, 2014. URL https://www.w3.org/TR/2014/REC-turtle-20140225/.
35. Hayes, P. and P. F. Patel-Schneider. *RDF 1.1 Semantics*. W3C recommendation, W3C, 2014. URL https://www.w3.org/TR/rdf11-mt/.
36. Arndt, D. and P.-A. Champin. *Notation3 Semantics*. W3C community group report, W3C, 2023. URL https://w3c.github.io/N3/reports/20230703/semantics.html.
37. Hassanzadeh, O. and M. P. Consens. Linked Movie Data Base. In *Proceedings of the Linked Data on the Web Workshop (LDOW 2009)*, *CEUR Workshop Proceedings*, Vol. 538. CEUR-WS.org, 2009. URL https://ceur-ws.org/Vol-538/ldow2009_paper12.pdf.
38. Pellissier Tanon, T., G. Weikum, and F. Suchanek. Yago 4: A Reason-able Knowledge Base. In *The Semantic Web: 17th International Conference, ESWC 2020, Proceedings*, *Lecture Notes in Computer Science*, Vol. 12123. Springer, 2020, pp. 583–596. doi:10.1007/978-3-030-49461-2_34.
39. HL7 International. HL7 Fast Health Interop Resources (FHIR). URL https://www.hl7.org/index.cfm.
40. U.S. National Library of Medicine. SNOMED CT. URL https://www.nlm.nih.gov/healthit/snomedct/index.html.
41. Bagan, G., A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 29, No. 4, 2017, pp. 856–869. doi:10.1109/TKDE.2016.2633993.