SQUALL: the expressiveness of SPARQL 1.1 made available as a controlled natural language [1]

Sébastien Ferré[a]

[a]IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes cedex, France

Abstract

The Semantic Web (SW) is now made of billions of triples, which are available as Linked Open Data (LOD) or as RDF stores. The SPARQL query language provides a very expressive way to search and explore this wealth of semantic data. However, user-friendly interfaces are needed to bridge the gap between end-users and SW formalisms. Navigation-based interfaces and natural language interfaces require no or little training, but they cover a small fragment of SPARQL's expressivity. We propose SQUALL, a query and update language that provides the full expressiveness of SPARQL 1.1 through a flexible controlled natural language (e.g., solution modifiers through superlatives, relational algebra through coordinations, filters through comparatives). A comprehensive and modular definition is given as a Montague grammar, and an evaluation of naturalness is done on the QALD challenge. SQUALL is conceived as a component of natural language interfaces, to be combined with lexicons, guided input, and contextual disambiguation. It is available as a Web service that translates SQUALL sentences to SPARQL, and submits them to SPARQL endpoints (e.g., DBpedia), therefore ensuring SW compliance, and leveraging the efficiency of SPARQL engines.

Keywords: Query language, Semantic Web, Expressiveness, Controlled natural language, SPARQL 1.1

1. Introduction

An open challenge of the Semantic Web [3] is *semantic search*, i.e., the ability for users to browse and search semantic data according to their needs. Semantic search systems can be classified according to their *usability*, the *expressive power* they offer, their *compliance* to Semantic Web standards, and their *scalability*. The most expressive approach by far is to use SPARQL [4], the standard RDF query language. SPARQL 1.1 [5] features graph patterns, filters, unions, differences, optionals, aggregations, expressions, subqueries, ordering, etc. However, SPARQL is also the least usable approach, because of the gap between users and the formal languages that RDF and SPARQL are. There

---

[1]This paper extends previous papers [1, 2] with substantial improvement of the SQUALL language, its presentation, and its evaluation.

Email address: `ferre@irisa.fr` (Sébastien Ferré)

are mainly two approaches to make semantic search more usable: navigation and natural language (NL). Navigation is used in *semantic browsers* (e.g., Fluidops Information Workbench[2]), and in *semantic faceted search* (e.g., SlashFacet [6], BrowseRDF [7], Sewelis [8]). Semantic faceted search can reach a significant expressiveness [8], but still much below SPARQL 1.1, and it does not scale easily to large datasets such as DBpedia[3]. Natural language is used in Ontology-based Query Answering (OQA) systems [9] in various forms, going from full natural language (e.g., FREyA [10], PowerAqua [11]) to mere keywords (e.g., NLP-Reduce [12]) through controlled natural languages (e.g., Ginseng [13]). Existing systems devote the most effort to bridging the gap between lexical forms and ontology triples (mapping and disambiguation), and process only the simplest questions, i.e., generate SPARQL queries with only one or two triples. Most of them support none of aggregations (e.g., counting), comparatives, or superlatives, even though those features are relatively frequent [14]. This means that even if full natural language is allowed as input, expressiveness is in fact strongly limited.

A less studied aspect is the update of RDF datasets, i.e., the insertion and deletion of triples. SPARQL 1.1 offers an update language to this purpose but with the same usability problem as the query language. Proposals for more usable interfaces have been made in faceted search (e.g., UTILIS [15]), and as a controlled natural language (e.g., ACE [16]). We think that update (and creation) of RDF data is as important as querying for end-users because it makes them first-class citizens, rather than consumers only.

In this paper, we define and evaluate SQUALL, a Semantic Query and Update High-Level Language[4]. Its contribution is: (1) to offer the full expressiveness of SPARQL 1.1 Query/Update (SPARQL for short) apart from a few details, (2) to cover a significant fragment of natural language (English), and (3) to be defined in a domain-independent way and in a concise way (its grammar has about 120 rules). SQUALL qualifies as a Controlled Natural Language (CNL) [17, 18] because it combines a fragment of natural language syntax, and the unambiguous semantics of formal languages. The main advantage of CNLs over formal languages is a better readability and understandability by people whose background knowledge does not cover logic or computer languages. SQUALL provides a lot of syntactic flexibility in that a same SPARQL query/update can be expressed in many different ways. To the best of our knowledge, no existing CNL target SPARQL queries and updates. Other CNLs for the Semantic Web rather target ontologies (e.g., ACE [19], SOS and Rabbit [20]). Because the focus of this paper is on syntactic and semantic expressiveness, we only assume a domain-independent basic default lexicon that uses qualified names (e.g., `dbo:Film`) as content words. In this setting, SQUALL is less natural at the lexical level, but applicable to SPARQL endpoints without any preparation. However, our approach makes it possible to define a customized lexicon (i.e., mapping words to possibly complex semantic forms), and is in principle compatible with mapping techniques used in OQA systems (i.e., using external resources such as ontologies and WordNet). SQUALL is also compatible with guided input (like in Gingseng [13]), which is recognized as important to solve the *habitability* problem in NL interfaces [12, 9].

---

[2] `http://iwb.fluidops.com/`

[3] `http://dbpedia.org`

[4] SQUALL's homepage at `http://www.irisa.fr/LIS/softwares/squall`.

SQUALL is available as two Web services[5]. A *translation form* takes a SQUALL sentence and returns its SPARQL translation. A *query form* takes a SPARQL endpoint URL, namespace definitions, and a SQUALL sentence, sends the SPARQL translation to the endpoint, which returns the list of query answers. The translation of SQUALL to SPARQL ensures compliance w.r.t. SW standards, and scalability by leveraging the efficiency of SPARQL engines.

Section 2 is a short introduction to Semantic Web formalisms (RDF and SPARQL). Section 3 gives an overview of the coverage of SPARQL features by SQUALL through examples. Section 4 develops a comprehensive definition of the syntax and semantics of SQUALL, where it is shown how each feature is covered by NL constructs. The result of syntactic parsing is a semantic intermediate representation, whose translation to SPARQL is addressed in Section 5. Section 6 evaluates SQUALL's expressiveness by defining a backward translation from each SPARQL construct to SQUALL. Section 7 evaluates the NL coverage, the naturalness, and the performance of SQUALL on questions from the QALD challenge (Query Answering over Linked Data) [14]. Finally, Section 8 compares SQUALL to related work, and Section 9 concludes and discusses perspectives.

2. Semantic Web: RDF and SPARQL

The Semantic Web (SW) is founded on several representation languages, such as RDF, RDFS, and OWL, which provide increasing inference capabilities [3]. The two basic units of these languages are *resources* and *triples*. A resource can be either a URI (Uniform Resource Identifier), a literal (e.g., a string, a number, a date), or a *blank node*, i.e., an anonymous resource. A URI is the absolute name of a *resource*, i.e., an entity, and plays the same role as a URL w.r.t. web pages. Like URLs, a URI can be a long and cumbersome string (e.g., `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`), so that it is often denoted by a qualified name, e.g., `rdf:type`, where `rdf:` is the RDF namespace. In the N3 notation[6], the default namespace `:` can be omitted for qualified names that do not collide with reserved keywords (*bare qualified names*).

A triple (*s p o*) is made of 3 resources, and can be read as a simple sentence, where *s* is the subject, *p* is the verb (called the predicate), and *o* is the object. For instance, the triple (`Bob knows Alice`) says that "Bob knows Alice", where `Bob` and `Alice` are the bare qualified names of two individuals, and `knows` is the bare qualified name of a property, i.e., a binary relation. The triple (`Bob rdf:type man`) says that "Bob has type man", or simply "Bob is a man". Here, the resource `man` is used as a class, and `rdf:type` is a property from the RDF namespace. The triple (`man rdfs:subClassOf person`) says that "man is a subclass of person", or simply "every man is a person". The set of all triples of a knowledge base forms an RDF graph.

RDF query languages [21] provide on semantic web knowledge bases the same service as SQL on relational databases. They generally assume that implicit triples have been inferred and added to the base. The standard RDF query language, SPARQL 1.1 [5], reuses the `SELECT FROM WHERE` shape of SQL queries, using graph patterns in the `WHERE` clause. A graph pattern *G* is one of:

- a triple pattern (s p o .) made of RDF terms and variables (e.g., `?x`),

- a join of two patterns (G_1 G_2),

- an union of two patterns (G_1 `UNION` G_2),

- a difference of two patterns (G_1 `MINUS` G_2),

- an optional pattern (`OPTIONAL` G_1),

- a filter pattern (`FILTER` C), where C is a constraint, i.e., a Boolean expression based on predicates (e.g., comparing, matching), functions (e.g., arithmetic, string concatenation), and the (un)satisfiability of a graph pattern (`(NOT) EXISTS` G_1),

- an assignment to a variable: either of the result of an expression (`BIND (`*expr* `AS` x`)`), or of a RDF resource (`VALUES` x `{` *res* `}`),

- a named graph pattern (`GRAPH` g G_1), where g denotes a named graph, in which the graph pattern G_1 should be matched,

- a service graph pattern (`SERVICE` *uri* G_1), where *uri* points to another SPARQL endpoint against which the graph pattern G_1 should be matched,

- a subquery.

Aggregations and expressions can be used in the `SELECT` clause (e.g., `COUNT(?x)`, `SUM(?x)`, `2 * ?x`), and `GROUP BY` clauses can be added to a query. Solution modifiers can also be added to the query for ordering results (`ORDER BY`) or returning a subset of results (`OFFSET`, `LIMIT`). Other query forms allow for closed questions (`ASK`), for returning the description of a resource (`DESCRIBE`), or for returning RDF graphs as results instead of tables (`CONSTRUCT`). SPARQL has been extended into an update language to insert/delete triples in/from a graph (`INSERT`, `DELETE`), and to manage RDF graphs (`LOAD`, `CLEAR`, etc.). The most general update form is `DELETE` D `INSERT` I `WHERE` G, where I and D can be sets of triple patterns plus named graph patterns, and G is a graph pattern that defines bindings for variables occuring in I and D.

3. SQUALL overview through SPARQL features

This section presents an overview of the SQUALL language by giving for each SPARQL feature its counterpart in SQUALL. It aims to give the reader a first taste of the language, and also a first assessment of its expressiveness compared to SPARQL. This list of SPARQL features is adapted and extended from a comparison of RDF query languages [21]. For each feature, SQUALL sentences are given as illustrations, with relevant parts underlined. For the sake of simplicity, we assume that all resources belong to a same namespace so that bare qualified names can be used (e.g., "person", "author", "Paper42"). The SPARQL translation of SQUALL sentences can be obtained from the translation form at `http://lisfs2008.irisa.fr/ocsigen/squall/`.

4

Triple patterns. Each noun or non-auxiliary verb plays the role of a class or predicate in a triple pattern. If a question is about the class or predicate itself, the verbs "belongs" and "relates" are respectively used.

- "Which <u>person</u> is the <u>author</u> of a <u>publication</u> whose <u>publication_year</u> is 2012?"

- "To which <u>nationality</u> does John_Smith <u>belong</u>?" (here, "nationality" is a meta-class whose instances are classes of persons: e.g., "French", "German").

- "What <u>relates</u> John_Smith to Mary_Well?"

Updates. Updates are obtained by declarative sentences. A sequence of declarative sentences generates a sequence of updates. Graph-level updates (e.g., `LOAD`, `CLEAR`) are obtained by imperative sentences.

- "Paper42 has author John_Smith and has publication_year 2012."

- "John_Smith know-s Mary_Well. Mary_Well know-s John_Smith."

- "<u>Load</u> <http://example.org/data.rdf> into Graph1."

- "<u>Clear</u> all named graphs."

Queries. `SELECT` queries are obtained by open questions, using one or several question words ("which" as a determiner, "what" or "who" as a noun phrase). Queries with a single selected variable can also be expressed as imperative sentences. `ASK` queries are obtained by closed questions, using either the word "whether" in front of a declarative sentence, or using auxiliary verbs and subject-auxiliary inversion.

- "<u>Which</u> person is the author of <u>which</u> publication?"

- "<u>Give me</u> the author-s of Paper42."

- "<u>Whether</u> John_Smith know-s Mary_Well?"

- "<u>Does Mary_Well</u> know the author of Paper42?"

Solution modifiers. The ordering of results (`ORDER BY`) and partial results (`LIMIT`, `OFFSET`) are expressed with superlatives.

- "Which person-s have the <u>10 greatest</u> age-s?"

- "Who are the author-s of the publication-s whose publication_year is the <u>2nd latest</u>?"

- "Which person is the author of <u>the most</u> publication-s?"

Join. The coordination "and" can be used with all kinds of phrases. It generates complex joins at the relational algebra level.

- "John_Smith <u>and</u> Mary_Well have age 42 <u>and</u> are an author of Paper42 <u>and</u> Paper43."

Union. Unions of graph patterns are expressed by the coordination "or", which can be used with all kinds of phrases, like "and".

- "Which teacher <u>or</u> student teach-es <u>or</u> attend-s a course whose topic is NL <u>or</u> DB?"

Option. Optional graph patterns are expressed by the adverb "maybe", which can be used in front of all kinds of phrases, generally verb phrases.

- "The author-s of Paper42 have which name and <u>maybe</u> have which email?"

Negation. The negative constraint on graph patterns (NOT EXISTS) is expressed by the adverb "not", which can be used in front of all kinds of phrases, and in combination with auxiliary verbs. In updates, negation entails the deletion of triples.

- "Which author of Paper42 <u>has not</u> affiliation Salford_University?"
- "John_Smith <u>is not</u> a teacher and <u>does not</u> teach Course101."

Quantification. Quantifiers have no direct counterpart in SPARQL, and can only be expressed indirectly with negation or aggregation. In SQUALL, they are expressed by determiners like "a", "every", "no", "some", "at least 3", "the". The latter "the" is interpreted existentially in queries, and universally in updates. The universal quantifier in updates allows for batches of updates, and corresponds to the use of a WHERE clause in SPARQL updates.

- "<u>Every</u> author of Paper42 has affiliation <u>the</u> university whose location is Salford."
- "Which publication has <u>more than 2</u> author-s whose affiliation is Salford_University?"

Built-ins. Built-in predicates and functions used in SPARQL filters and expressions are expressed by pre-defined nouns, verbs, and relational adjectives. They can therefore be used like classes and properties.

- "Which person has a birth_date whose <u>month</u> is 3 and whose <u>year</u> is <u>greater than</u> 2000?"
- "Give me the publication-s whose title <u>contains</u> "natural language"?"

Expressions. Operators and functions are defined as coordinations so that they can be applied on different kinds of phrases: e.g., relational nouns, noun phrases.

- "Which publication has <u>the lastPage - the firstPage + 1</u> greater than 10?" (page number)
- "Return <u>concat(the firstname, " ", the lastname)</u> of all author-s of Paper42." (fullname)

Aggregation and grouping. Aggregation is expressed by the question determiner "how many", by relational nouns such as "sum", and by adjectives such as "total", "average". Grouping clauses are introduced by the word "per".

- "<u>How many</u> publication-s have author John_Smith?"
- "What is the <u>number of</u> publication-s <u>per</u> author?"
- "What is the <u>average</u> age of the author-s of a publication <u>per</u> affiliation?"

Property paths. Property sequences and inverse properties are covered by the flexible syntax of SQUALL. Alternative and negative paths are respectively covered by the coordination "or" and the adverb "not". Reflexive and transitive closures of properties have no obvious linguistic counterpart, and are expressed so far by property suffixes among "?", "+", and "*". SQUALL does not yet support the transitive closure of complex property paths (e.g., (^author/author)+ for co-authors of co-authors, etc.).

- "Which publication-s cite+ Paper42?" (i.e., *Which publications cite Paper42 or cite a publication that cites Paper42, etc?*)

Named graphs. The GRAPH (resp. SERVICE) construct of SPARQL, which serves to restrict graph pattern solutions to a named graph (resp. to a distant service), can be expressed using "in graph" (resp. "from service") as a preposition. A prepositional phrase can be inserted at any location in a sentence, and its scope is the whole sentence.

- "Who is the author of the most publication-s in graph Salford_Publications?"

- "In which graph is John_Smith the author of at least 10 publication-s?"

- "What is the dbpedia-owl:capital of dbpedia:France from service <http://dbpedia.org/>?"

Graph literals. The SPARQL query forms CONSTRUCT and DESCRIBE return graphs, i.e. sets of triples, instead of sets of solutions. A DESCRIBE query is expressed by the imperative verb "describe" followed by a resource or a universally-quantified noun phrase. A CONSTRUCT query is expressed by using a subordinate clause introduced by "that", which is reified into a graph literal.

- "Describe the author-s of Paper42."

- "For every publication with an author X and with an author Y different from X, return that X has coauthor Y and Y has coauthor X."

Collection patterns. SPARQL has a special notation for collection literals (e.g., (1 2 3)) but not for collection patterns (e.g., Paper42 has an author list whose last element is John Smith). In SPARQL, collection patterns are expressed by combining triple patterns, blank nodes, and property paths: e.g., :Paper42 :authorList [rdf:rest* [rdf:first :John_Smith ; rdf:nil]]. SQUALL offers concise and powerful patterns by reusing Prolog's notations plus the ellipsis, and translating them into SPARQL.

- "What has authorList [..., John_Smith]?"

- "Paper42 has authorList [..., who]?" (i.e., *Who is the last author of Paper42?*)

- "Paper42 has authorList [..., who, ...]?" (i.e., *Who are the authors of Paper42?*)

- "Paper42 has authorList [_, who, ...]?" (i.e., *Who is the second author of Paper42?*)

4. Syntax and semantics

In this section, we formally define the syntax and semantics of SQUALL in the style of Montague grammars. Montague grammars [22] are an approach to natural language semantics that is based on formal logic and λ-calculus. It is named after the American logician Richard Montague, who pioneered this approach [23]. A Montague grammar is a context-free generative grammar, where each rule is decorated by a λ-expression that denotes the semantics of the syntactic construct defined by the rule. The semantics is defined in a fully compositional style, i.e., the semantics of a construct is always a composition of the semantics of sub-constructs. The obtained semantics for a valid SQUALL sentence is represented in an intermediate logical language, rather than directly in terms of an existing query language for the Semantic Web. This is a common practice in the compilation of high-level programming languages, and has a number of advantages. First, it makes the semantics easier to write and understand because defined at a more abstract level. As a side effect, it would also make it easier to redefine SQUALL for other natural languages than English. Second, it gives freedom in the choice of the implementation. For instance, the operational semantics of the intermediate language can be given by translating it to an existing language, e.g., SPARQL; by interpreting it in a relational algebra engine; or by using continuation passing-style, like in Prolog. In Section 5, we describe a solution for the first approach.

SQUALL sentences are decomposed into noun phrases, verb phrases, relatives, determiners, prepositional phrases, etc. As an illustration, we consider a complex sentence that covers many features of SQUALL: "For which researcher-s X, in graph DBLP every publication whose author is X and whose publication_year is greater than 2000 has at least 2 author-s?". Its syntactic analysis is

$$[_S\text{for } [_{NP}[_{Det}\text{which}] \ [_{NG1}[_{Noun1}\text{researcher-s}] \ [_{App}[_{Label}\text{X}]]]],$$
$$[_S[_{PP}[_{PrepNoun}\text{in graph}] \ [_{NP}[_{ProperNoun}\text{DBLP}]]]$$
$$[_S[_{NP}[_{Det}\text{every}] \ [_{NG1}[_{Noun1}\text{publication}]$$
$$[_{Rel}[_{Rel}\text{whose } [_{NG2}[_{Noun2}\text{author}]] \ [_{VP}\text{is } [_{NP}[_{Label}\text{X}]]]]$$
$$\text{and } [_{Rel}\text{whose } [_{NG2}[_{Noun2}\text{publication\_year}]]$$
$$[_{VP}\text{is } [_{Rel}\text{greater than } [_{NP}[_{Literal}\text{2000}]]]]]]]]]]$$
$$[_{VP}\text{has } [_{Det}\text{at least 2}] \ [_{Noun2}\text{author-s}]]]]],$$

and its semantic intermediate representation is

$$\textbf{select } \lambda r.(\textbf{triple } r \ \texttt{rdf:type :researcher})$$
$$\lambda r.(\textbf{context } \texttt{GRAPH :DBLP} \ (\textbf{forall}$$
$$\lambda p.(\textbf{exists } \lambda y.(\textbf{and}$$
$$(\textbf{triple } p \ \texttt{rdf:type :publication}) \ (\textbf{triple } p \ \texttt{:author } r)$$
$$(\textbf{triple } p \ \texttt{:publication\_year } y) \ (\textbf{pred } > y \ 2000)))$$
$$\lambda p.(\textbf{exists } \lambda n.(\textbf{and}$$
$$(\textbf{aggreg } \texttt{COUNT} \ \lambda[].\lambda a.(\textbf{triple } p \ \texttt{:author } a) \ n)$$
$$(\textbf{pred } >= n \ 2)))))).$$

In the following, we successively define the semantic types and expressions (Section 4.1) used for the intermediate representation, notations for Montague grammars (Section 4.2), SQUALL's lexical units and our default lexicon (Section 4.3), and

type	definition	variable names	name of the type
e	-	x, y, z	entity
s	-	s	statement
$p1$	$e \rightarrow s$	d	description
$p2$	$e \rightarrow e \rightarrow s$	r	relation
$s1$	$p1 \rightarrow s$	q	quantifier
$s2$	$p1 \rightarrow p1 \rightarrow s$	$q2$	binary quantifier
m_α	$\alpha \rightarrow \alpha$	m	α-modifier
c_α	$\alpha \rightarrow \alpha \rightarrow \alpha$	c	α-coordination

Table 1: List of semantic types along with their definition (except for base types), the common name of their variables, and a short description of the type.

SQUALL's syntactic rules (Section 4.4). We also explain how syntactic ambiguities are resolved (Section 4.5), how non-local aspects are handled by semantic transformations (Section 4.6), and how the intermediate representation is semantically validated (Section 4.7).

4.1. Semantic types and expressions

Montague grammar are based on *simply-typed λ-calculus* [24]. Every syntagm is associated to some semantic type, and those types constrain the way semantic expressions can be combined. The two base types are e for *entities*, and s for *statements*. The main type constructor is $\alpha \rightarrow \beta$ for functions from expressions of type α to expressions of type β. The sub-types α and β can themselves be function types, recursively. For instance, an expression of type $p1 = e \rightarrow s$ expects an entity, and returns a statement: it can be seen as a statement missing an entity. For example, the verb phrase "knows Mary" has semantic type $p1$ because it misses an entity (e.g., "John") as the subject so as to make a complete statement. Table 1 lists and defines the semantic types that are used in this paper.

There are two kinds of semantic expressions associated to function types: *applications* and *abstractions*. The application of an expression f of type $\alpha \rightarrow \beta$ to an expression e of type β is noted $f\ e$, and has type β. The abstraction of an expression e of type β by a variable x of type α is noted $\lambda x.e$, and has type $\alpha \rightarrow \beta$. In the notation of expressions, abstraction has priority over application, and application is left-associative: e.g., $e_1\ \lambda x.e_2\ e_3 = ((e_1\ (\lambda x.e_2))\ e_3)$. Expressions obtained by composition can be simplified according to λ-calculus, through β-reduction ($(\lambda x.s)\ y =_\beta s[x \leftarrow y]$, where $s[x \leftarrow y]$ denotes the substitution of x by y in s), and η-expansion ($d =_\eta \lambda x.(d\ x)$, if d is a function).

For convenience, we also introduce a type constructor for lists, i.e., $[\alpha]$ for lists whose elements have type α. In semantic expressions, $[x; y; z]$ denotes a list with 3 elements, $[]$ is the empty list, and $(x :: l)$ is a list whose first element is x, and whose rest is l[7].

Constants in semantic expressions play the role of semantic constructors in the intermediate representation of SQUALL sentences. Table 2 lists all the necessary constants, called *primitives*, to define the semantics of SQUALL. Although they take their name

---

[7]The reader may have recognized the notations from ML [25].

primitive	type	description
triple	$e \rightarrow e \rightarrow e \rightarrow s$	triple pattern
command	$n \rightarrow [e] \rightarrow s$	command
pred	$n \rightarrow [e] \rightarrow s$	predicate call
func	$n \rightarrow [e] \rightarrow p1$	function call
aggreg	$n \rightarrow ([e] \rightarrow p1) \rightarrow p1$	aggregation
modif	$n \rightarrow m_s$	solution modifier (e.g., ordering)
context	$n \rightarrow e \rightarrow m_s$	context (e.g., named graph, service)
arg	$n \rightarrow e \rightarrow m_s$	defining extra-argument
true	s	tautology
not	m_s	negation
option	m_s	optional
and	c_s	conjunction
or	c_s	disjunction
implies	c_s	implication
where	c_s	conditional
exists	$s1$	existential quantifier
forall	$s2$	universal quantifier
the	$s2$	definite quantifier
forarg	$n \rightarrow s1$	using extra-argument
ask	m_s	query constructor
select	$s1$	interrogative quantifier
select_where	$s2$	binary interrogative quantifier
return	$p1$	interrogative imperative
label	$p2$	entity labelling by a user variable
ref	$e \rightarrow e$	entity reference by a user variable
graphliteral	$s \rightarrow e$	graph reification as entity

Table 2: List of semantic primitives, along with their type and description.

from English, they are in fact determined by SPARQL features, and independent to the source natural language. In types, the base type n is introduced as a sub-type of e for primitive arguments that correspond to names (e.g., predicates, functions, solution modifiers, extra-arguments). The table is given here in full for reference, but the meaning of each primitive is only explained on their first introduction in the Montague grammar. For convenience, we define additional constants on top of those primitives for comparators and counting:

$\quad$ **eq** $: p2 = \lambda x.\lambda y.(\textbf{pred} = [x; y])$
$\quad$ **gt** $: p2 = \lambda x.\lambda y.(\textbf{pred} > [x; y])$
$\quad$ **between** $: e \rightarrow p2 = \lambda x.\lambda y_1.\lambda y_2.(\textbf{and} \ (\textbf{pred} >= [x; y_1]) \ (\textbf{pred} <= [x; y_2]))$
$\quad$ **count** $: p1 \rightarrow p1 = \lambda d.\lambda n.(\textbf{aggreg} \ \texttt{COUNT} \ \lambda[].\lambda y.(d \ y) \ n)$

4.2. Notations for Montague grammars

$\quad$ A Montague grammar is made of rules. Each rule has a syntactic part, and a semantic part. The syntactic part is a generative rule (e.g., $S \rightarrow$ **there is** *NP*), where grammatical words are in bold, and syntagms are in italic. The semantic part is a λ-expression

between curly brackets (e.g., { np $\lambda x.$**true** }), where constants are in bold, RDF terms are in teletype, and variables are in italic. For each syntagm (e.g., NP) in the right-hand side of the generative rule, its semantics is denoted by a variable whose name is a lowercase prefix of the syntagm name (e.g., np). Indices are used to disambiguate between several occurences of a same syntagm (e.g., np_1, np_2). For brevity, alternative words or syntagms are combined with a slash (e.g., **is/are**), instead of writing several rules. Each syntagm is associated to a single semantic type, which is specified in the first rule defining it (e.g., $S : s$). A default semantic can be defined at the same time (e.g., $Rel : p1$ (default : $\lambda x.$**true**)), and allows to make the syntagm optional in right-hand sides of rules (e.g., $Rel?$). In trivial rules like $X \rightarrow Y$ { y }, the semantic part may be left implicit. To support modularity and avoid duplication of rules, we use higher-order syntagms, i.e., syntagms parametrized by other syntagms. There are two parametrization modes: $X(Y)$ is used when the semantics of Y is incorporated into the semantics of X, and $X\{Y\}$ is used when the semantics of Y is not part of X, and is returned apart. The latter is useful to parse non-contiguous constructs (e.g., "**between** ... **and** ... "). A beauty of Montague grammars is that they can be directly encoded in functional programming languages because they are based on lambda-calculus. Therefore, the grammar detailed in the following sections constitutes an executable specification of SQUALL's semantics.

4.3. Lexical units and default lexicon

Table 3 lists the different lexical units that are used in SQUALL's grammar, along with their semantic type. That list could be different for other natural languages than English, although there would probably be a large overlap. An entity can be any of a proper noun, a literal, a number, and a label. A description (type $p1$) can be any of a common noun, an adjective, an intransitive verb, and an imperative verb. A relation can be any of a relation noun, a transitive adjective, and a transitive verb. A statement modifier can be any of a preposition, and a superlative. Functions and operators return a description of their result given a list of entities (arguments). Aggregators return a description of their result (the aggregated value) given a description of what has to be aggregated ($p1 \rightarrow p1$). However, when there are dimensions (e.g., a `GROUP BY` clause), the description of what has to be aggregated depends on the values taken on each dimension ($[e]$), hence the type ($[e] \rightarrow p1) \rightarrow p1$ for aggregators. This type is a complex analogue of the type of quantifiers ($p1 \rightarrow s$).

Because the focus of this paper is about the SQUALL language and its correspondence with SPARQL, we do not elaborate much about the lexical analysis, i.e., about the translation from concrete word forms to a Semantic Web vocabulary. There is active research on this task (see Section 8), and integrating their results is left to future work. The current implementation of SQUALL has a default definition of lexical units based only on URIs, and SPARQL built-ins (see below), but the way it is coded makes it relatively easy to extend those definitions for a particular vocabulary. Just to give an exemple, assuming a data property `ex:color`, the adjective "green" can be defined by the following rule:

$$Adj1 \rightarrow \textbf{green} \ \{ \ \lambda x.(\textbf{triple } x \ \texttt{ex:color } "green") \ \}$$

This rule says that a "green" thing is a thing that has color `"green"`.

By default, all URIs can be used as a proper noun (e.g., `res:France`); class URIs can be used as common nouns (e.g., `:Person`); property URIs can be used as relation

syntagm	type	name (examples)
ProperNoun	e	proper noun (John, France)
Literal	e	literal value (2013, "Hello")
Number	e	number (3, 0.5)
Label	e	label (X, Y2, ?foo)
Noun1	$p1$	common noun (man, country)
Noun2	$p2$	relation noun (mother, birth date)
Adj1	$p1$	adjective (green, French)
Adj2	$p2$	transitive adjective (knowing, known by)
Verb1	$p1$	intransitive verb (works, drink)
Verb2	$p2$	transitive verb (knows, wrote)
VerbImp	$p1$	imperative verb (load, print)
QueryImp	$p1$	imperative question prefix (return, give me)
Func	$[e] \rightarrow p1$	function (sqrt, pgcd)
Nulop	$p1$	nullary operator (now, a random number)
Unop	$e \rightarrow p1$	unary prefix operator (-)
Mulop	$e \rightarrow e \rightarrow p1$	×-priority infix operator (*, /)
Addop	$e \rightarrow e \rightarrow p1$	+-priority infix operator (+, -)
NounAggreg	$([e] \rightarrow p1) \rightarrow p1$	aggregation noun (count, sum)
AdjAggreg	$([e] \rightarrow p1) \rightarrow p1$	aggregation adjective (total, average)
Prep	$e \rightarrow m_s$	preposition (to, from)
PrepNoun{Det}	$e \rightarrow m_s$	preposition (in {a} graph, from {the} service)
AdjSuper	$e \rightarrow m_s$	superlative adjective (10 greatest, 2nd)
DetSuper	$e \rightarrow m_s$	superlative determiner (the most, the 2nd most)

Table 3: List of the lexical units along with their semantic type, name, and concrete examples.

nouns (e.g., `:author`) and transitive verbs (e.g., `:livesIn`), and function URIs can be used as functions. Absolute URIs, relative URIs, and qualified names are written like in SPARQL. In the default namespace, bare qualified names can also be used, like in notation N3 (e.g., `author` instead of `:author`). Qualified names allow for relatively natural sentences, as shown in examples in this paper and on the Web page.

$$ProperNoun \rightarrow URI \quad \{ \; uri \; \}$$
$$Noun1 \rightarrow URI \quad \{ \; \lambda x.(\textbf{triple} \; x \; \texttt{rdf:type} \; uri) \; \}$$
$$Noun2/Verb2 \rightarrow URI \quad \{ \; \lambda x.\lambda y.(\textbf{triple} \; x \; uri \; y) \; \}$$
$$Func \rightarrow URI \quad \{ \; \lambda lx.\lambda y.(\textbf{func} \; uri \; lx \; y) \; \}$$

In SPARQL triple patterns, a variable can be used in place of a class or a property. To support this feature, a triple `?x rdf:type ?c` is expressed as "X belongs to C"; and a triple `?x ?p ?y` is expressed as "P relates X to Y". "belong(s)" is a predefined intransitive verb, and "relate(s)" is a predefined transitive verb. They both rely on the predefined preposition "to" to specify the object of the triple (see Section 4.4.6).

$$Verb1 \rightarrow \textbf{belong(s)} \quad \{ \; \lambda x.(\textbf{forarg} \; \texttt{to} \; \lambda c.(\textbf{triple} \; x \; \texttt{rdf:type} \; c)) \; \}$$
$$Verb2 \rightarrow \textbf{relate(s)} \quad \{ \; \lambda p.\lambda x.(\textbf{forarg} \; \texttt{to} \; \lambda y.(\textbf{triple} \; x \; p \; y)) \; \}$$

SPARQL built-ins can be used under various forms, according to their semantic type. Unary predicates can be used as common nouns (e.g., "literal" for `isLiteral(.)`), adjectives (e.g., "numeric" for `isNumeric(.)`), and intransitive verbs. Binary predicates can be used as relation noun (e.g., "prefix" for `strStarts(.,.)`), transitive adjective (e.g., "matching" for `REGEX(.,.)`), and transitive verbs (e.g., "contain(s)" for `contains(.,.)`). All SPARQL functions can be used as SQUALL functions. Unary functions can also be used like binary predicates (e.g., "length" for `strlen(.)`), and nullary functions can be used as nullary operators (e.g., "a random number" for `RAND()`). Aggregations can be used as aggregation nouns and adjectives (e.g., the adjective "total" for `SUM`). SPARQL commands like `DESCRIBE` and `LOAD` are expressed as imperative verbs (resp., "describe", "load"). The SPARQL constructs `GRAPH` and `SERVICE` are expressed by the noun prepositions "in *Det* graph" and "from *Det* service". Indeed, the latter constructs modify a statement given an entity (either a named graph or a service). The other default prepositions are "into" and "to" to be used in the commands `LOAD` and `ADD/MOVE/COPY`. The SPARQL solution modifiers combine the ordering of solutions (`ORDER BY`), and the selection of a range of solutions (`LIMIT` and `OFFSET`). They are expressed as superlative adjectives and determiners. For example, the adjective highest translates to the SPARQL modifier `ORDER BY DESC(`x`) LIMIT 1`, where x is the entity passed to highest. We finally give a few representative examples about defining SPARQL built-ins as lexical units.

$Noun1 \rightarrow$ **literal** { $\lambda x.$(**pred** `isLiteral` $[x]$) }
$AdjAggreg \rightarrow$ **total** { $\lambda d.\lambda x.$(**aggreg** `SUM` d x) }
$VerbImp \rightarrow$ **load** { $\lambda y.$(**forarg** into $\lambda z.$(**command** `LOAD` $[y; z]$)) }
$AdjSuper \rightarrow$ **highest** { $\lambda x.\lambda s.$(**modif** "`ORDER BY DESC(`x`) LIMIT 1`" s) }
$PrepNoun\{Det\} \rightarrow$ **in** Det **graph** { $\lambda x.\lambda s.$(**context** `GRAPH` x s) }

4.4. Syntactic rules

In this section, we describe the syntactic rules of SQUALL in a modular fashion, where each module corresponds either to some SPARQL feature, or to some natural language feature, and often to a combination of both. This has the benefits to split the whole grammar (about 120 rules) in smaller parts, and to emphasize the relationships between SQUALL and SPARQL. This is also a way to assess the coverage of SPARQL by SQUALL (see Section 6 for a more rigorous evaluation of expressiveness). The reader should feel free to skip some modules, or to ignore the formal grammars on first reading.

4.4.1. Triples as sentences

A triple (s p o) can be seen as a basic sentence. The tradition in linguistics [26] is to analyse s and o as noun phrases (NP), p o as a verb phrase (VP), and the whole triple as a sentence (S). Here, noun phrases are either proper nouns or literal values, but they are given the type $s1$ to prepare for the use of quantifiers in Section 4.4.2. Verb phrases are based either on an intransitive verb (*Verb1*) followed by an optional complement phrase (*CP*), or on a transitive verb (*Verb2*) followed by an object phrase (*OP*), which is a noun phrase followed by an optional complement phrase. Complements are defined in Section 4.4.6, and are assumed empty at this point.

$$S \; : \; s$$
$$\quad \to NP \; VP \; \{ \; np \; vp \; \}$$
$$NP \; : \; s1$$
$$\quad \to Term \; \{ \; \lambda d.(d \; term) \; \}$$
$$Term \; : \; e$$
$$\quad \to ProperNoun$$
$$\quad \to Literal$$
$$VP \; : \; p1$$
$$\quad \to VPdo$$
$$VPdo \; : \; p1$$
$$\quad \to Verb1 \; CP? \; \{ \; \lambda x.(cp \; (vp \; x)) \; \}$$
$$\quad \to Verb2 \; OP \; \{ \; \lambda x.(op \; \lambda y.(verb2 \; x \; y)) \; \}$$
$$OP \; : \; s1$$
$$\quad \to NP \; CP? \; \{ \; \lambda d.(np \; \lambda y.(cp \; (d \; y))) \; \}$$
$$CP \; : \; m_s \quad (default : \lambda s.s)$$

4.4.2. Quantifiers as determiners

Quantifiers are commonplace in natural languages in the form of determiners, whereas they are notoriously difficult to express in SPARQL or SQL [27]. Determiners behave as binary quantifiers (type $s2$), but unary quantifiers (type $s1$) can be cast as binary quantifiers by conjuncting the two descriptions. The definite article **the** has its own semantic primitive because it is interpreted either as the existential quantifier or as the universal quantifier depending on its syntactic context (see Section 4.6). A noun phrase can be formed by a determiner followed by a noun group (*NG1*). A binary noun phrase (*NP2*) is a noun phrase abstracted over an entity, and is made of a relation noun group (*NG2*). Noun groups are based on a noun (a relational noun for *NG2*), and may be modified by an adjective, an apposition (*App*), and a relative clause (*Rel*) (to be defined in the following sections). The keywords **for** and **there** introduce global quantifiers, in a style close to mathematical logic.

The grammar rules are defined so that the scope of quantifiers are leftmost-outermost, and are restricted to the scope of the related verb. Therefore, "every man love-s some woman" means there is possibly a different woman for each man; while "there is a woman that every man love-s" means there is a single woman that is loved.

Det : $s2$
 $\rightarrow Det1$ { $\lambda d_1.\lambda d_2.(det1\ \lambda x.(\textbf{and}\ (d_1\ x)\ (d_2\ x)))$ }
 $\rightarrow$ **every/all** { $\lambda d_1.\lambda d_2.(\textbf{forall}\ d_1\ d_2)$ }
 $\rightarrow$ **the** { $\lambda d_1.\lambda d_2.(\textbf{the}\ d_1\ d_2)$ }
$Det1$: $s1$
 $\rightarrow$ **a/an** { $\lambda d.(\textbf{exists}\ d)$ }
 $\rightarrow$ **some** { $\lambda d.(\textbf{exists}\ d)$ }
 $\rightarrow$ **no** { $\lambda d.(\textbf{not}\ (\textbf{exists}\ d))$ }
$NP \rightarrow Det\ NG1$ { $\lambda d.(det\ ng1\ d)$ }
 $\rightarrow NP2$ **of** NP { $\lambda d.(np\ \lambda x.(np2\ x\ d))$ }
$NP2$: $e \rightarrow s1$
 $\rightarrow Det\ NG2$ { $\lambda x.\lambda d.(det\ \lambda y.(ng2\ x\ y)\ d)$ }
$NG1$: $p1$
 $\rightarrow Adj1?\ Noun1\ App?\ Rel?$ { $\lambda x.(\textbf{and}\ (app\ x)\ (noun1\ x)\ (adj1\ x)\ (rel\ x))$ }
$NG2$: $p2$
 $\rightarrow Adj1?\ Noun2\ App?$ { $\lambda x.\lambda y.(\textbf{and}\ (app\ y)\ (noun2\ x\ y)\ (adj1\ y))$ }
S $\rightarrow$ **for** NP, S { $np\ \lambda x.s$ }
 $\rightarrow$ **there is/are/was/were** NP { $np\ \lambda x.\textbf{true}$ }

4.4.3. Labels as appositions and anaphoras

 Labels are used to cover the phenomenon of anaphoras in natural languages. They are introduced as appositions into noun groups, and referenced as terms. References are resolved at the semantic level, taking into account the scope of variables to check their validity, and to resolve ambiguities (see Section 4.7). Labels are the analogues of variables in SPARQL queries, but are much less often necessary in the syntax of a natural language.

$$App\ :\ p1\quad (\text{default} : \lambda x.\textbf{true})$$
$$\rightarrow Label\ \{\ \lambda x.(\textbf{label}\ x\ label)\ \}$$
$$Term \rightarrow Label\ \{\ \textbf{ref}\ label\ \}$$

4.4.4. Relative clauses

 Relative clauses modify the nouns, and semantically, are statements abstracted over an entity. Therefore, they can be derived from the syntax and semantics of sentences, replacing a noun phrase by a keyword among **that/which/who/whom**, and moving it at the beginning of the relative clause. Relative clauses can also be similar to a verb phrase, only using adjectives (including participles) instead of verbs.

Rel : $p1$ (default : $\lambda x.\textbf{true}$)
  $\rightarrow$ **that/which/who** VP { $\lambda x.(vp\ x)$ }
  $\rightarrow$ **that/which/whom** NP $Verb2$ $CP?$ { $\lambda y.(np\ \lambda x.(cp\ (verb2\ x\ y)))$ }
  $\rightarrow$ $NP2$ **of which** VP { $\lambda x.(np2\ x\ vp)$ }
  $\rightarrow$ **whose** $NG2$ VP { $\lambda x.(\textbf{exists}\ \lambda y.(\textbf{and}\ (ng2\ x\ y)\ (vp\ y)))$ }
  $\rightarrow$ $Adj1$ $CP?$ { $\lambda x.(cp\ (adj1\ x))$ }
  $\rightarrow$ $Adj2$ OP { $\lambda x.(op\ \lambda y.(adj2\ x\ y))$ }
  $\rightarrow$ **such that** S { $\lambda x.s$ }

4.4.5. Auxiliary verbs

A particular syntax and semantics is associated to the auxiliary verbs "to be" and "to have", and verb phrases take three forms accordingly. Auxiliaries can be negated, and they are given the semantics of a sentence modifier, either negation or identity. The verb "to be" can be followed by nothing (keyword **there**), a relative clause (mostly adjective forms), and a *copula noun phrase*. Copula noun phrases (*NPC* and *NPC2*) have a syntax similar to noun phrases (*NP* and *NP2*), but a different semantics (type $p1$ instead of $s1$). They are parametrized by a semantic-less determiner syntagm, here an article. Copula noun phrases are also used in queries (Section 4.4.7) and aggregations (Section 4.4.12).

The verb "to have" can be followed by a relation noun (playing the role of a transitive verb) and an object phrase (e.g., "John_Smith has spouse Mary_Well"), or by a relational noun group and an optional relative clause (e.g., "John_Smith has every child that is a doctor"). The prepositions "with/without" play the same role as "to have", but for relative clauses instead of verb phrases (e.g., "every man with a child").

$Aux(Root)$: m_s
  $\rightarrow$ $Root$ { $\lambda s.s$ }
  $\rightarrow$ $Root$ **not** { $\lambda s.(\textbf{not}\ s)$ }
  $\rightarrow$ $Root\textbf{n}'\textbf{t}$ { $\lambda s.(\textbf{not}\ s)$ }
$VP \rightarrow Aux(\textbf{does/do/did})$ $VPdo$ { $\lambda x.(aux\ (vp\ x))$ }
  $\rightarrow$ $Aux(\textbf{is/are/was/were})$ $VPbe$ { $\lambda x.(aux\ (vp\ x))$ }
  $\rightarrow$ $Aux(\textbf{has/have/had})$ $VPhave$ { $\lambda x.(aux\ (vp\ x))$ }

16

$$VPbe \ : \ p1$$
$$\rightarrow \textbf{there} \ \{ \ \lambda x.\textbf{true} \ \}$$
$$\rightarrow Rel \ \{ \ \lambda x.(rel \ x) \ \}$$
$$\rightarrow NPC(\textbf{a/an/the}) \ \{ \ \lambda x.(npc \ x) \ \}$$
$$NPC(Det) \ : \ p1$$
$$\rightarrow Term \ \{ \ \lambda x.(\textbf{eq} \ x \ term) \ \}$$
$$\rightarrow Det \ NG1 \ \{ \ \lambda x.(ng1 \ x) \ \}$$
$$\rightarrow NPC2(Det) \ \textbf{of} \ NP \ \{ \ \lambda y.(np \ \lambda x.(npc2 \ x \ y)) \ \}$$
$$NPC2(Det) \ : \ e \rightarrow p1$$
$$\rightarrow Det \ NG2 \ \{ \ \lambda x.\lambda y.(ng2 \ x \ y) \ \}$$
$$VPhave \ : \ p1$$
$$\rightarrow Noun2 \ OP \ \{ \ \lambda x.(op \ \lambda y.(noun2 \ x \ y)) \ \}$$
$$\rightarrow NP2 \ Rel? \ \{ \ \lambda x.(np2 \ x \ rel) \ \}$$
$$Rel \rightarrow \textbf{with} \ VPhave \ \{ \ \lambda x.(vp \ x) \ \}$$
$$\rightarrow \textbf{without} \ VPhave \ \{ \ \lambda x.(\textbf{not} \ (vp \ x)) \ \}$$

4.4.6. Prepositional phrases

Prepositional phrases (PP) are used to handle verbs that expect arguments in addition to subject and object (e.g., "John belongs <u>to</u> the `rdfs:Class` that has `rdfs:label` "human""), and also to express *truth-contexts* such as the named graphs and distant services of SPARQL (e.g., "<u>from</u> the graph whose author is John   Paris is the capital of France"). A prepositional phrase is made of a preposition and a noun phrase. The noun phrase is used like subjects and objects, and the preposition modifies the statement in function of the variable z introduced by the noun phrase (e.g., passing the extra-argument to the verb, see Section 4.6). In the case of a noun preposition ($PrepNoun$), the noun of the noun phrase helps to determine the semantics of the preposition (e.g., "from {the} graph"). A propositional phrase can occur at any position in a sentence: at the beginning or at the end of the sentence, before or after the verb. The nesting of S, VP, OP, and CP ensures the leftmost-outermost rule for the scope of quantifiers, and the free position of complements allows for their flexible ordering. Finally, as a relative clause is a sentence with a displaced noun phrase, a form of relative clause is generated for each form of prepositional phrase (e.g., "to which John belongs", "in which graph Paris is the capital of France").

$$S \rightarrow PP \ S \ \{ \ pp \ s \ \}$$
$$VP \rightarrow PP \ VP \ \{ \ \lambda x.(pp \ (vp \ x)) \ \}$$
$$OP \rightarrow PP \ OP \ \{ \ \lambda d.(pp \ (op \ d)) \ \}$$
$$CP \rightarrow PP \ CP? \ \{ \ \lambda s.(pp \ (cp \ s)) \ \}$$
$$PP \ : \ m_s$$
$$\rightarrow Prep \ NP \ \{ \ \lambda s.(np \ \lambda z.(prep \ z \ s)) \ \}$$
$$\rightarrow PrepNoun\{Det\} \ App? \ Rel? \ \{ \ \lambda s.(det \ \lambda z.(\textbf{and} \ (app \ z) \ (rel \ z)) \ \lambda z.(prep \ z \ s)) \ \}$$
$$Rel \rightarrow Prep \ \textbf{which} \ S \ \{ \ \lambda x.(prep \ x \ s) \ \}$$
$$\rightarrow PrepNoun\{\textbf{which}\} \ S \ \{ \ \lambda x.(prep \ x \ s) \ \}$$

17

4.4.7. Updates and queries

Updates (U) are either a sentence ended by a full-stop (e.g., "John knows Mary."), or an imperative verb followed by an object phrase and a full-stop (e.g., "Clear all named graphs."), or a sequence of updates. Queries (Q) can be closed or open. Closed queries can be formed by ending a declarative sentence by a question mark (e.g., "John knows Mary?"), possibly prefixing it with the keyword **whether**. The semantic primitive **ask** modifies the statement into an interrogative one. Open queries are expressed either with imperative locutions (e.g., return, give me) followed by an object phrase that describes what has to be returned (e.g., "Give me all film-s whose director is Tim_Burton."), or with the interrogative words **which/what/who/whom**. The semantic primitive **return** specifies the variable to be projected in results. The semantic primitives **select** and **select_where** quantify over a variable to be projected (the SELECT clause in SPARQL). A multi-dimensional query is a sentence with several occurences of wh-words (e.g., "Which person is the director of which film?").

$$
\begin{aligned}
U \;:\; & s \\
& \to S \,.\; \{\, s \,\} \\
& \to VerbImp \; OP \,.\; \{\, op\; \lambda x.(verbimp\; x) \,\} \\
& \to U\; U \;\; \{\, \textbf{and}\; u_1\; u_2 \,\} \\
Q \;:\; & s \\
& \to S \,?\; \{\, \textbf{ask}\; s \,\} \\
& \to \textbf{whether}\; S \,?\; \{\, \textbf{ask}\; s \,\} \\
& \to QueryImp \; OP \,.\; \{\, op\; \lambda x.(\textbf{return}\; x) \,\} \\
NP \to & \textbf{what/who/whom} \;\; \{\, \lambda d.(\textbf{select}\; d) \,\} \\
Det \to & \textbf{which} \;\; \{\, \lambda d_1.\lambda d_2.(\textbf{select\_where}\; d_1\; d_2) \,\}
\end{aligned}
$$

To allow for a more natural verbalization of closed questions, the grammar has been extended to support subject-auxiliary inversion (e.g., "Does John know Mary?", "Is John the father of a doctor?"). Those grammar rules can be derived from previous rules, and we do not detail them here.

4.4.8. Solution modifiers as superlatives

SPARQL solution modifiers include ordering results w.r.t. a variable or expression (ORDER BY), and selecting a sub-range of solutions (LIMIT and OFFSET). In SQUALL, they are expressed as superlative adjectives (e.g., highest like in "Which mountain has the highest elevation?"), and they modify a statement in function of the entity the adjective applies to. Those adjectives are special in that they must occur first and only once in noun groups. See Section 4.6 for a deeper explanation of their semantics.

$$
\begin{aligned}
NG1 \to & AdjSuper \; NG1 \;\; \{\, \lambda x.(adj\; x\; (ng1\; x)) \,\} \\
NG2 \to & AdjSuper \; NG2 \;\; \{\, \lambda x.\lambda y.(adj\; y\; (ng2\; x\; y)) \,\}
\end{aligned}
$$

4.4.9. Relational algebra as coordinations

There is a one-to-one mapping between operations from the relational algebra of SPARQL (join, union, optional, negation), coordinations (resp., and, or, not, maybe), and

Boolean operators that we use in our intermediate representation. We define Boolean expressions ($Bool(X)$) in a generic way by parametrizing them with atomic expressions X. In this way, they can be defined once, and applied to many syntagms. For example, the grammar rule $S \rightarrow Bool(S)$ means that sentences can be coordinated (e.g., "John own-s a book X and Mary own-s X."). In SQUALL, all the following syntagms can be fully coordinated: S, NP, $NP2$, Rel, VP, OP, CP, PP, $Prep$, $Noun1$, $Noun2$, $Adj1$, $Adj2$, $Verb1$, and $Verb2$. This helps to make sentences more concise by factorizing common phrases (e.g., "Do John and Mary own or rent every book whose topic is "CNL"?"). Parentheses can be used to override the usual priorities between Boolean operators, but they are hardly ever necessary in practice. Because Boolean expressions are applied to syntagms having different types α, the Boolean semantic primitives must be extended from statements to those types. For example, $(\textbf{and}_{p1}\ d_1\ d_2) = \lambda x.(\textbf{and}\ (d_1\ x)\ (d_2\ x))$, and $(\textbf{and}_{p2}\ r_1\ r_2) = \lambda x.\lambda y.(\textbf{and}\ (r_1\ x\ y)\ (r_2\ x\ y))$. More generally, every additional argument is passed down to sub-expressions.

$$
\begin{aligned}
Bool(X : \alpha)\ &:\ \alpha \\
&\rightarrow Bool(X)\ \textbf{or}\ Bool(X)\ \{\ \textbf{or}_\alpha\ bool_1\ bool_2\ \} \\
&\rightarrow Bool(X)\ \textbf{and}\ Bool(X)\ \{\ \textbf{and}_\alpha\ bool_1\ bool_2\ \} \\
&\rightarrow \textbf{maybe}\ Bool(X)\ \{\ \textbf{option}_\alpha\ bool\ \} \\
&\rightarrow \textbf{not}\ Bool(X)\ \{\ \textbf{not}_\alpha\ bool\ \} \\
&\rightarrow (\ Bool(X)\ )\ \{\ bool\ \} \\
&\rightarrow X\ \{\ x\ \}
\end{aligned}
$$

4.4.10. Conditionals

Conditionals are introduced by the keywords **if-then**, and are semantically defined with implications (e.g., "If John has a spouse X then X is a woman."). The coordination **where** is interpreted as a reverse implication in updates, and as a conjunction in queries (e.g., "Which film F has a director D where D is an actor of F?").

$$
\begin{aligned}
S\ &\rightarrow \textbf{if}\ S\ \textbf{then}\ S\ \{\ \textbf{implies}\ s_1\ s_2\ \} \\
&\rightarrow S\ \textbf{where}\ S\ \{\ \textbf{where}\ s_1\ s_2\ \}
\end{aligned}
$$

4.4.11. Expressions

In SPARQL, expressions are found in filters, in the `SELECT` clause, and in solution modifiers (`ORDER BY`, `HAVING`). They combine operators, functions, literals, and variables. In SQUALL, expressions ($Expr(X)$) are defined in a generic way by parametrizing them with a syntagm for atomic expressions X, like for Boolean expressions. This allows to use expressions in all kinds of noun phrases (see below). Syntactically, expressions are classically made of atomic expressions (X), terms (include literals and references), additive and multiplicative infix operators, prefix unary operators, nullary operators (e.g., the SPARQL function `NOW`), functions, and parentheses. Semantically, an expression has type $k_\alpha = (e \rightarrow \alpha) \rightarrow \alpha$, i.e., is defined in Continuation Passing Style (CPS). In short, each expression takes a *continuation* as an argument, i.e., a function that says what to do with the result of the evaluation of the expression. The definition of k_α makes expressions analogue to unary quantifiers, and hence to noun phrases. Indeed, the SQUALL sentence

"The height * the width of some rectangle R is 100" is equivalent to "For the height H of some rectangle R, for the width W of R, H * W is 100". The definition of the semantics of expressions rely on a combinator *apply* that applies a continuation k to the result of an operation or function.

$$apply : (e \to \alpha) \to p1 \to \alpha = \lambda k.\lambda d.\lambda z^*.(\textbf{the } \lambda y.(d\ y)\ \lambda y.(k\ y\ z^*))$$

The number of extra arguments z^* is equal to the arity of type α. The quantifier **the** ensures that expressions will be put in the WHERE-clause of updates.

$Expr(X : k_\alpha)\ :\ k_\alpha$
 $\to Expr(X)\ Addop\ Expr(X)$
  $\{\ \lambda k.(expr_1\ \lambda x_1.(expr_2\ \lambda x_2.(apply\ k\ (addop\ x_1\ x_2))))\ \}$
 $\to Expr(X)\ Mulop\ Expr(X)$
  $\{\ \lambda k.(expr_1\ \lambda x_1.(expr_2\ \lambda x_2.(apply\ k\ (mulop\ x_1\ x_2))))\ \}$
 $\to Unop\ Expr(X)\ \{\ \lambda k.(expr\ \lambda x.(apply\ k\ (unop\ x)))\ \}$
 $\to Nulop\ \{\ \lambda k.(apply\ k\ nulop)\ \}$
 $\to Func(Expr(X),...,Expr(X))$
  $\{\ \lambda k.(expr_1\ \lambda x_1.(...\ expr_n\ \lambda x_n.(apply\ k\ (func\ [x_1;\ldots;x_n]))\ ...))\ \}$
 $\to (\ Expr(X)\ )\ \{\ expr\ \}$
 $\to Term\ \{\ \lambda k.(k\ term)\ \}$
 $\to X\ \{\ x\ \}$

Expressions apply to all kinds of noun phrases: *NP*, *NP2*, *NPC*, and *NPC2*. Because expressions and the different noun phrases have different types, noun phrases need to be wrapped as atomic expressions, and whole expressions need to be unwrapped back as noun phrases.

$NP \to Expr(NP\ \{\ \lambda k.\lambda d.(np\ \lambda x.(k\ x\ d))\ \})$
 $\{\ expr\ \lambda v.\lambda d.(d\ v)\ \}$
$NP2 \to Expr(NP2\ \{\ \lambda k.\lambda x.\lambda d.(np2\ x\ \lambda y.(k\ y\ x\ d))\ \})$
 $\{\ expr\ \lambda v.\lambda x.\lambda d.(d\ v)\ \}$
$NPC(Det) \to Expr(NPC(Det)\ \{\ \lambda k.\lambda x.(\textbf{exists}\ \lambda y.(\textbf{and}\ (npc\ y)\ (k\ y)))\ \})$
 $\{\ \lambda x.(expr\ \lambda v.(\textbf{eq}\ x\ v))\ \}$
$NPC2(Det) \to Expr(NPC2(Det)\ \{\ \lambda k.\lambda x.(\textbf{exists}\ \lambda y.(\textbf{and}\ (npc2\ x\ y)\ (k\ y)))\ \})$
 $\{\ \lambda x.\lambda y.(expr\ \lambda v.(\textbf{eq}\ y\ v)\ x)\ \}$

4.4.12. Aggregations and grouping

Aggregations are another way to compute values, beside expressions. However, they behave differently w.r.t. syntax and semantics because an aggregator applies to a description (i.e., a set of values). Moreover, one or several dimensions can be specified (GROUP BY clauses in SPARQL) so that an aggregation may return a set of aggregated values (type $p1$ instead of e). For instance, "the average size of the person-s per age" returns an average size for each instanciated age. Hence the type $g = ([e] \to p1) \to p1$ of aggregators, where $[e]$ represents the list of dimension variables, the first $p1$ represents the

set of values to be aggregated, and the second $p1$ represents the set of aggregated values. Aggregators can be seen as quantifiers, which quantify existentially over each dimension. In fact, an adjective aggregator can be used as a determiner, where the aggregated values are constrained by an optional relative clause (e.g., "John has a number greater than 3 of paper-s"). Alternately, aggregators can be the head of noun groups so that aggregated values can be used as subjects and objects of verbs (e.g., "Is every number of publication-s per author lesser than 100?"), and so that aggregations can be nested (e.g., "What is the average number of publication-s per author?"). Each dimension is defined as a relation from the *facts* (e.g., "publications" in previous examples) to the dimension variable z. A list of dimensions is a relation from the facts to a list lz of dimension variables.

$Det1 \rightarrow$ **a number** $App?$ $Rel?$ **of**
      $\{ \lambda d.(\textbf{the } \lambda x.(\textbf{count } d\ x)\ \lambda x.(\textbf{and } (app\ x)\ (rel\ x))) \}$
   $\rightarrow$ **a/an** $AdjAggreg$ $App?$ $Rel?$
      $\{ \lambda d.(\textbf{the } \lambda x.(aggreg\ \lambda lz.\lambda y.(d\ y)\ x)\ \lambda x.(\textbf{and } (app\ x)\ (rel\ x))) \}$
$NG1 \rightarrow NounAggreg$ $App?$ **of** $NPC(\textbf{the})$ $Dims?$
      $\{ \lambda v.(\textbf{and } (app\ v)\ (aggreg\ \lambda lz.\lambda y.(\textbf{and } (npc\ y)\ (dims\ y\ lz))\ v)) \}$
   $\rightarrow AdjAggreg$ $App?$ $NG1$ $Dims?$
      $\{ \lambda v.(\textbf{and } (app\ v)\ (aggreg\ \lambda lz.\lambda y.(\textbf{and } (ng1\ y)\ (dims\ y\ lz))\ v)) \}$
   $\rightarrow AdjAggreg$ $App?$ $NG2$ **of** $NPC(\textbf{the})$ $Dims?$
      $\{ \lambda v.(\textbf{and } (app\ v)\ (aggreg\ \lambda lz.\lambda y.(\textbf{exists } \lambda x.($
            and $(npc\ x)\ (ng2\ x\ y)\ (dims\ x\ lz)))\ v)) \}$
$Dims\ :\ e \rightarrow [e] \rightarrow s$ (default : $\lambda y.\lambda[].\textbf{true}$)
   $\rightarrow Dim$ $\{ \lambda y.\lambda[z].(dim\ y\ z) \}$
   $\rightarrow Dim$ **and** $Dims$ $\{ \lambda y.\lambda(z::lz).(\textbf{and } (dim\ y\ z)\ (dims\ y\ lz)) \}$
$Dim\ :\ p2$
   $\rightarrow$ **per** $NG2$ $\{ \lambda y.\lambda z.(ng2\ y\ z) \}$
   $\rightarrow$ **per** $NPC(\textbf{the})$ $\{ \lambda y.\lambda z.(npc\ z) \}$

4.4.13. Comparisons

   Although comparators have type $p2$ and can therefore be verbalized as transitive verbs (e.g., is greater than), or as transitive adjectives (e.g., greater than), they deserve a special treatment because of the rich combinations offered by natural languages. We define below a number of constructs that do not increase SQUALL's expressivity, but that improves a lot SQUALL's naturalness and concision when expressing comparisons. We only detail here the "greater than" comparator (constant **gt**), but other comparators are straightforward to add.

   Relatives are extended by using comparators as transitive adjectives, and by adding "between" as a ditransitive adjective. Their semantics is the same as for transitive adjectives.

$Rel \rightarrow$ **greater than** OP $\{ \lambda x.(op\ \lambda y.(\textbf{gt } x\ y)) \}$
   $\rightarrow$ **between** OP **and** OP $\{ \lambda x.(op_1\ \lambda y_1.(op_2\ \lambda y_2.(\textbf{between } x\ y_1\ y_2))) \}$

Comparators are also used as determiners after the auxiliary verb "to have" to compare two properties of an entity (e.g., "Which rectangle has a greater height than width?"), or to compare a same property between two entities (e.g., "Which woman has a greater size than every man?").

$DetComp\{X, Y\}$: $s2$
    $\rightarrow$ **a greater** X **than** Y
        $\{ \lambda d_1.\lambda d_2.(\mathbf{exists}\ \lambda y_1.(\mathbf{exists}\ \lambda y_2.(\mathbf{and}\ (d_1\ y_1)\ (d_2\ y_2)\ (\mathbf{gt}\ y_1\ y_2)))) \}$
$VPhave$
    $\rightarrow DetComp\{Noun2, Noun2\}$
        $\{ \lambda x.(detcomp\ \lambda y_1.(noun2_1\ x\ y_1)\ \lambda y_2.(noun2_2\ x\ y_2)) \}$
    $\rightarrow DetComp\{Noun2, NP\}$
        $\{ \lambda x_1.(np\ \lambda x_2.(detcomp\ \lambda y_1.(noun2\ x_1\ y_1)\ \lambda y_2.(noun2\ x_2\ y_2))) \}$

There are also determiners to compare numbers of things, where the values to be compared are the result of a COUNT-aggregation. The semantic statement $(\mathbf{count}\ d\ n)$ tells that n is the number of entities satisfying description d. It is used to define generalized determiners such as "more than 2" or "between 3 and 5", the interrogative determiner "how many", and superlative determiners such as "the most". From these, it becomes possible to express queries such: "How many person-s are the author of at least 10 publication-s?" or "Which person is the author of the most publication-s?".

$Det1 \rightarrow Number$ $\{ \lambda d.(\mathbf{count}\ d\ number) \}$
    $\rightarrow$ **more than** $Number$ $\{ \lambda d.(\mathbf{exists}\ \lambda n.(\mathbf{and}\ (\mathbf{count}\ d\ n)\ (\mathbf{gt}\ n\ number))) \}$
    $\rightarrow$ **between** $Number$ **and** $Number$
        $\{ \lambda d.(\mathbf{exists}\ \lambda n.(\mathbf{and}\ (\mathbf{count}\ d\ n)\ (\mathbf{between}\ n\ number_1\ number_2))) \}$
    $\rightarrow$ **how many** $\{ \lambda d.(\mathbf{select}\ \lambda n.(\mathbf{count}\ d\ n)) \}$
    $\rightarrow DetModif$ $\{ \lambda d.(\mathbf{exists}\ \lambda n.(modif\ n\ (\mathbf{count}\ d\ n))) \}$

We also define binary determiners to compare two numbers of things, which lead to new forms of noun phrases:

- "Do <u>more woman than man</u> own a cat?"

- "Who has <u>more daughter-s than son-s</u> whose age is lesser than 18?"

- "Which woman X has <u>more daughter-s than the mother of X</u>?"

    $DetCompCount\{X, Y\}$: $s2$
        $\rightarrow$ **more** X **than** Y $\{ \lambda d_1.\lambda d_2.(\mathbf{exists}\ \lambda n_1.(\mathbf{exists}\ \lambda n_2.($
            $\mathbf{and}\ (\mathbf{count}\ d_1\ n_1)\ (\mathbf{count}\ d_2\ n_2)\ (\mathbf{gt}\ n_1\ n_2)))) \}$
    $NP \rightarrow DetCompCount\{NG1, NG1\}$ $\{ \lambda d.(detcomp$
      $\lambda x_1.(\mathbf{and}\ (ng1_1\ x_1)\ (d\ x_1))\ \lambda x_2.(\mathbf{and}\ (ng1_2\ x_2)\ (d\ x_2))) \}$
    $NP2 \rightarrow DetCompCount\{NG2, NG2\}$ $\{ \lambda x.\lambda d.(detcomp$
      $\lambda y_1.(\mathbf{and}\ (ng2_1\ x\ y_1)\ (d\ y_1))\ \lambda y_2.(\mathbf{and}\ (ng2_2\ x\ y_2)\ (d\ y_2))) \}$
        $\rightarrow DetCompCount\{NG2, NP\}$ $\{ \lambda x_1.\lambda d.(np\ \lambda x_2.(detcomp$
      $\lambda y_1.(\mathbf{and}\ (ng2\ x_1\ y_1)\ (d\ y_1))\ \lambda y_2.(\mathbf{and}\ (ng2\ x_2\ y_2)\ (d\ y_2)))) \}$

4.4.14. Graph literals

In SPARQL, `CONSTRUCT`-queries return graphs, i.e., sets of triples. In order to support this kind of query in SQUALL, we allow the reification of statements as entities using the primitive **graphliteral** in the semantics, and the word **that** followed by a sentence in the syntax to introduce a subordinate clause. The reification is invalid if the statement is anything else than a conjunction of triples, and if any variable remains unbound. Like in notation N3, graph literals (called formulae in N3) can be used everywhere an entity can be used. However, when translating to SPARQL, they can only be returned as the query result (e.g., "For every person X whose age is greater or equal to 18, return that X is an adult.").

$$Term \rightarrow \textbf{that } S \quad \{ \textbf{ graphliteral } s \}$$

4.4.15. Collections

RDF collections are single-chained lists based on properties `rdf:first` and `rdf:rest`, plus the empty list `rdf:nil`. They are useful to represent closed and ordered sets of entities, such as the authors of a document. Turtle and SPARQL provide syntactic sugar for fixed-length collections, but the representation of other graph patterns is extremely tedious. For example, the SPARQL pattern to access the last element of a collection is: `[ rdf:rest* [ rdf:first ?last ; rdf:rest rdf:nil ] ]`. We extend SQUALL's noun phrases so as to completely avoid the use of the RDF vocabulary about collections. We reuse the rich Prolog's syntax for list patterns [28], and extend it with the ellipsis ... to allow jumping over an arbitrary number of elements. List patterns are delimited by square brackets, list elements are separated by commas, and sublists are raised after a bar. An underscore _ stands for an arbitrary element (joker), and an ellipsis ... stands for an arbitrary sequence of elements. For example, "Which book has authorList [..., John_Smith]?" returns the books whose last author is John Smith, and "Whether [_, every member of Team1, ...] is the authorList of a book whose topic is AI?" returns whether every member of Team1 is the second author of a book whose topic is AI.

List noun phrases (*NPL*) are noun phrases that quantify over lists. To simplify the definition of their semantics, we rely on two combinators *cons* and *sublist* which build unary quantifiers (type *s1*), respectively, from a first and a rest, and from a sublist:

$cons = \lambda e.\lambda l.\lambda d.(\textbf{exists } \lambda x.(\textbf{and } (\textbf{triple } x \texttt{ rdf:first } e) \: (\textbf{triple } x \texttt{ rdf:rest } l) \: (d \; x)))$
$sublist = \lambda l.\lambda d.(\textbf{exists } \lambda x.(\textbf{and } (\textbf{triple } x \texttt{ rdf:rest* } l) \: (d \; x)))$

$$
\begin{aligned}
Term &\rightarrow [\,] \quad \{ \texttt{ rdf:nil } \} \\
NP &\rightarrow \_ \quad \{ \lambda d.(\textbf{exists } d) \} \\
&\rightarrow [\; NPL\;] \quad \{ \; npl \; \} \\
NPL \; &: \; s1 \\
&\rightarrow NP, \; NPL \quad \{ \lambda d.(np \; \lambda e.(npl \; \lambda l.(cons \; e \; l \; d))) \} \\
&\rightarrow NP \mid NP \quad \{ \lambda d.(np_1 \; \lambda e.(np_2 \; \lambda l.(cons \; e \; l \; d))) \} \\
&\rightarrow NP \quad \{ \lambda d.(np \; \lambda e.(cons \; e \; \texttt{ rdf:nil } d)) \} \\
&\rightarrow ..., \; NPL \quad \{ \lambda d.(npl \; \lambda l.(sublist \; l \; d)) \} \\
&\rightarrow ... \mid NP \quad \{ \lambda d.(np \; \lambda l.(sublist \; l \; d)) \} \\
&\rightarrow ... \quad \{ \lambda d.(sublist \; \texttt{ rdf:nil } d) \}
\end{aligned}
$$

4.5. Handling of syntactic ambiguities

The price for the natural and flexible syntax of SQUALL is ambiguity, i.e., the fact that some sentences can be parsed in different ways possibly leading to different semantics. In SQUALL, ambiguities are resolved by the following rules:

1. when forming a construct X from one or two constructs of same syntagm X (e.g., coordinating 2 NPs, modifying a sentence with a PP), algebraic operators have priority (in decreasing priority order: **not**, **maybe**, **and**, **or**, **where**) over sentence modifiers (PP as a prefix, and global quantifiers **for** NP), and right-associativity applies for binary coordinations;
2. "smaller" syntagms have priority over "larger" syntagms, i.e., in decreasing priority order: $Det*$, Rel, $NG*$, $NP*$, PP, CP, OP, VP, S;
3. in case of remaining ambiguity between two phrases of the same syntagm, the shorter phrase is chosen.

Round brackets can be used in coordinations and expressions to escape those rules. Rule 2 implies that "a man or woman" is interpreted as "a (man or woman)" rather than "(a man) or woman", as $NG1$ has priority over NP. Rule 3 implies that in "A know-s a researcher that X cite-s in graph G", the PP "in graph G" binds to the shorter VP "cite-s ..." rather than to the longer VP "know-s ...".

4.6. Semantic transformations of statements

Semantic transformations need to be applied to the intermediate representation produced through Montague grammars, because some primitives are contextual in nature, i.e., have non-local effects. This concerns the interrogative quantifiers (primitives **select** and **select_where**), prepositions (primitive **arg**), and modifiers (primitive **modif**). For example, the SQUALL sentence "Every member of Group1 belongs to which nationality?" (equivalent to "To which nationality does every member of Group1 belong?"), gets the following intermediate representation (some lexical units have not been expanded by their definition for clarity):

$$\textbf{forall } \lambda x.(member \ x \ \texttt{:Group1})$$
$$\lambda x.(\textbf{select\_where } \lambda z.(nationality \ z) \ \lambda z.(\textbf{arg to } z \ (belong \ x)))$$

Two problems appear in this example. First, the primitive **select_where** should be out of the scope of the quantifier **forall**. Second, the sub-expression $(belong \ x)$ should reduce to $(\textbf{triple } x \ \texttt{rdf:type } z)$, but z is not an argument of $belong$.

The solution we have found to retain a compositional style (i.e., to avoid global variables), while maintaining a local treatment of each feature, is to define statements as *state monads* [29], i.e. as functions that take a state as a parameter, and that return a modified state in addition to the statement itself. The additional state enables to pass information downward, and upward in the semantic expression tree. The state is here made of extra-arguments added by prepositions, selectors added by interrogative quantifiers, and modifiers. Selectors are passed upward up to the root of the syntax tree. Modifiers are also passed upward, but are catched by the determiner of noun phrases (NP), to account for sentences such as "Does the person with the lowest age has a greater size than the person with the highest age?". Prepositions are passed downward with primitive **arg** and catched by verbs with primitive **forarg** (e.g., the preposition "to" is

catched by the verbs "belongs" and "relates"). Applying those principles to the above example, we get the fully reduced intermediate representation.

$$\textbf{select } \lambda z.(\textbf{and } (\textbf{triple } z \text{ rdf:type :nationality})$$
$$(\textbf{forall } \lambda x.(\textbf{triple } x \text{ :member :Group1}) \ \lambda x.(\textbf{triple } x \text{ rdf:type } z)))$$

Note that **select_where** has been transformed into **select**, and has now the outermost scope. Note also that primitives **arg** and **forarg** have been eliminated in the transformed intermediate representation. Similarly, primitive **the** (resp. **where**) is replaced by primitive **exists** (resp. **and**) in a query context (in the scope of **ask**, **select**, and the left argument of **forall** and **implies**); and is replaced by **forall** (resp. the inverse of **implies**) otherwise.

4.7. Semantic validation

Like in programming languages, SQUALL sentences may be syntactically correct, but semantically invalid. Examples of semantic errors in programming languages are undeclared variables, or type errors. In SQUALL, semantic errors can be:

- out-of-scope references: e.g. the reference X in "John has no child X and X̲ is a doctor.";

- unbound variables: e.g. "Return 1 + some thing.", where there is no way to bind the second operand of the addition.

In order to detect semantic errors, and return them to users, statements are validated w.r.t. accessibility and boundedness. References are first resolved in function of defined labels, and accessible variables, which eliminates primitives **label** and **ref** from the intermediate representation. Accessibility validation consists in checking that every variable is used in the scope of its quantifier. Discourse Representation Theory (DRT) [30], and its combination with Montague grammars [31], is used to define those scopes, and plays an important role in the naturalness of SQUALL sentences. For example, this validation phase accepts the sentence "Some man X loves a woman and X is a doctor.", and rejects "Every man X loves a woman and X is a doctor.". Boundedness validation consists in checking that dataflow constraints are satisfied. For example, all arguments of predicates and functions must be bound before evaluating them, while triple patterns have no such constraints. This validation phase ensures that queries and updates can effectively be evaluated, i.e., are semantically well-defined. For example, the boundedness validation accepts "Which person has every child that is a doctor?", and rejects "Who has every child that is a doctor?" because the latter does not provide any means to bind the variable quantified by Who[8]. Boundedness validation is also used to replace equalities (**pred** $= [x; y]$) with the identity function when only one argument is bound (e.g., **func** ID $[x]$ y, if only x is bound).

5. Translation to SPARQL

The generation of a SPARQL query or update from the intermediate representation of semantics is much simpler than syntactic and semantic analysis because it mostly consists in mapping logical constructs to SPARQL constructs, which are at the same

---

[8]Unless a default class for people is specified, and used in the semantics of who.

level of abstraction. Note that the intermediate representation would make it easy to support another target RDF query language. As an illustration, the SPARQL translation of the SQUALL sentence "For which researcher-s X, in graph DBLP every publication whose author is X and whose :publication_year is greater than 2000 has at least 2 author-s?", from the introduction of Section 4, is as follows:

```
SELECT DISTINCT ?x1 WHERE {
    ?x1 a :researcher .
    FILTER NOT EXISTS {
        GRAPH :DBLP {
            ?x2 a :publication .
            ?x2 :author ?x1 .
            ?x2 :year ?x3 .
            FILTER (?x3 > 2000) . }
        FILTER NOT EXISTS {
            GRAPH :DBLP {
                { SELECT DISTINCT ?x2 (COUNT(?x5) AS ?x4)
                  WHERE { ?x2 :author ?x5 . }
                  GROUP BY ?x2 }
                FILTER (?x4 >= 2) . } } } }
```

The two nested `FILTER NOT EXISTS` encode the universal quantifier "every", and the subquery with aggregation encodes the numeric quantifier "at least 2".

The fully-reduced intermediate representation, after semantic transformations and validation (Sections 4.6 and 4.7), is only made of the following semantic primitives: **triple**, **pred**, **func**, **aggreg**, **modif**, **command**, **context**, **true**, **not**, **and**, **or**, **option**, **implies**, **exists**, **forall**, **ask**, **select**, **return**, **graphliteral** (see Table 2). We define in the following a mapping of those primitives to SPARQL 1.1 constructs. The notation $[\![s]\!]$ denotes the SPARQL translation of the statement s. Such a mapping provides at the same time a concrete semantics for SQUALL, and a practical way to evaluate SQUALL sentences by leveraging existing SPARQL engines and endpoints. We first consider the translation of interrogative sentences, and then the translation of imperative and declarative sentences. While the former always generate SPARQL queries, the latter generate both queries and updates. In all cases, the translation is not designed to produce optimal SPARQL code, but to be simple. This should not be a high concern given that SPARQL engines perform optimizations before evaluating queries and updates.

5.1. Interrogative sentences

Interrogative sentences have intermediate representations that start with primitives **ask** or **select**, which respectively generate `ASK`-queries and `SELECT`-queries in SPARQL. The notation $[\![X \mid s]\!]_Q$ denotes an auxiliary translation for multidimensional queries (nested **select**) where X is a row of variables, and the notation $[\![s]\!]_G$ denotes the translation of a statement into a graph pattern. Every occurence of a SPARQL variable $?x$ assumes the generation of a fresh variable name. Those variables are used to instantiate the description parameters of quantifiers and aggregations.

$$[\![\textbf{ask } s]\!] = \text{ASK } \{ \; [\![s]\!]_G \; \} \qquad\qquad [\![X \mid \textbf{select } d]\!]_Q = [\![X \; ?x \mid d \; ?x]\!]_Q$$
$$[\![\textbf{select } d]\!] = [\![?x \mid d \; ?x]\!]_Q \qquad\qquad [\![X \mid s]\!]_Q = \text{SELECT } X \text{ WHERE } \{ \; [\![s]\!]_G \; \}$$

Most statements can be mapped to graph patterns. Predicates and functions translate to SPARQL filters, and aggregations translate to SPARQL aggregative sub-queries. Modifiers translate to solution modifiers using sub-queries. Contexts translate to named graph patterns, and service patterns. Algebraic constructors translate to their SPARQL counterpart, and quantifiers all translate to the implicit SPARQL existential quantifier and negation.

$$[\![\textbf{triple } s\ p\ o]\!]_G = s\ \ p\ \ o\ \ .$$
$$[\![\textbf{pred } p\ lx]\!]_G = \texttt{FILTER } p(lx)$$
$$[\![\textbf{func ID } [x]\ y]\!]_G = \texttt{BIND } (x \texttt{ AS } y)$$
$$[\![\textbf{func } f\ lx\ y]\!]_G = \texttt{BIND } (f(lx) \texttt{ AS } y)$$
$$[\![\textbf{aggreg } g\ d\ x]\!]_G =$$
$$\quad \{ \texttt{ SELECT } ?z^* \ (g(?y) \texttt{ AS } x)$$
$$\qquad \texttt{WHERE } \{ \ [\![d\ ?z^*\ ?y]\!]_G \ \}$$
$$\qquad \texttt{GROUP BY } ?z^* \ \}$$
$$[\![\textbf{modif } m\ s]\!] = \{ \texttt{ SELECT * WHERE } \{ \ [\![s]\!]_G \ \} \ m \ \}$$
$$[\![\textbf{context GRAPH } x\ s]\!]_G = \texttt{GRAPH } x \ \{ \ [\![s]\!]_G \ \}$$
$$[\![\textbf{context SERVICE } x\ s]\!]_G = \texttt{SERVICE } x \ \{ \ [\![s]\!]_G \ \}$$
$$[\![\textbf{true}]\!]_G = \epsilon$$
$$[\![\textbf{and } s_1\ s_2]\!]_G = [\![s_1]\!]_G \ [\![s_2]\!]_G$$
$$[\![\textbf{or } s_1\ s_2]\!]_G = \{ \ [\![s_1]\!]_G \ \} \texttt{ UNION } \{ \ [\![s_2]\!]_G \ \}$$
$$[\![\textbf{option } s]\!]_G = \texttt{OPTIONAL } \{ \ [\![s]\!]_G \ \}$$
$$[\![\textbf{not } s]\!]_G = \texttt{FILTER NOT EXISTS } \{ \ [\![s]\!]_G \ \}$$
$$[\![\textbf{exists } d]\!]_G = [\![d\ ?x]\!]_G$$
$$[\![\textbf{forall } d_1\ d_2]\!]_G = [\![\textbf{not } (\textbf{exists } \lambda x.(\textbf{and } (d_1\ x)\ (\textbf{not } (d_2\ x))))]\!]_G$$

5.2. Imperative and declarative sentences

In imperative and declarative sentences, universal quantifiers (**forall**) and implications (**implies**) introduce a WHERE-clause, and existential quantifiers introduce new blank nodes (using SPARQL function BNODE()). In order to simplify the following mappings, we first define a few simplification rules that reduce all quantifiers to implications, and that collapse nested implications.

$$\textbf{exists } d = \textbf{forall } \lambda x.(\textbf{func BNODE } []\ x)\ d$$
$$\textbf{forall } d_1\ d_2 = \textbf{implies } (d_1\ ?x)\ (d_2\ ?x)$$
$$\textbf{implies } s_1\ (\textbf{implies } s_2\ s_3) = \textbf{implies } (\textbf{and } s_1\ s_2)\ s_3$$

Declarative sentences translate to updates that add and/or delete triples. The SPARQL form depends on the presence of a WHERE-clause.

$$[\![\textbf{implies } s_1\ s_2]\!] = \texttt{DELETE } \{ \ D \ \} \texttt{ INSERT } \{ \ I \ \} \texttt{ WHERE } \{ \ [\![s_1]\!]_G \ \}$$
$$\qquad \text{given } [\![\texttt{DEFAULT} : s_2]\!]_U = (I, D)$$
$$[\![s]\!] = \texttt{INSERT DATA } \{ \ I \ \}; \texttt{ DELETE DATA } \{ \ D \ \}$$
$$\qquad \text{given } [\![\texttt{DEFAULT} : s]\!]_U = (I, D)$$

This translation relies on the auxiliary translation $[\![g : s]\!]_U$ that generates two graph patterns (I, D), where I contains the triples to be inserted, and D contains the triples to

be deleted. The parameter g specifies in which graph triples should be inserted/removed.

$\llbracket \texttt{DEFAULT} : \textbf{triple } s\ p\ o \rrbracket_U = (s\ \ p\ \ o\ \ .,\epsilon)$
$\llbracket g : \textbf{triple } s\ p\ o \rrbracket_U = (\texttt{GRAPH}\ g\ \{\ s\ \ p\ \ o\ \ .\},\epsilon)$
$\llbracket g : \textbf{and } s_1\ s_2 \rrbracket_U = (I_1\ I_2, D_1\ D_2)$ given $\llbracket g : s_1 \rrbracket_U = (I_1, D_1), \llbracket g : s_2 \rrbracket_U = (I_2, D_2)$
$\llbracket g : \textbf{not } s \rrbracket_U = (D, I)$ given $\llbracket g : s \rrbracket_U = (I, D)$
$\llbracket g : \textbf{context GRAPH } x\ s \rrbracket_U = \llbracket x : s \rrbracket_U$

Imperative sentences can generate many forms of SPARQL queries and updates, depending on the imperative verb. Only some of those forms accept a `WHERE`-clause, and the mapping is therefore a partial function. For example, the command `CLEAR` cannot be used with a `WHERE`-clause, even though it would make sense: e.g., "Clear all named graphs in which graph John owns a unicorn.".

$\llbracket \textbf{command } \texttt{LOAD}\ [x; y] \rrbracket = \texttt{LOAD}\ x\ \texttt{INTO GRAPH}\ y$
$\llbracket \textbf{command } \texttt{CLEAR}\ [x] \rrbracket = \texttt{CLEAR}\ x$
$\llbracket \textbf{command } \texttt{DESCRIBE}\ [x] \rrbracket = \texttt{DESCRIBE}\ x$
$\llbracket \textbf{implies } s_1\ (\textbf{command } \texttt{DESCRIBE}\ [x]) \rrbracket = \texttt{DESCRIBE}\ x\ \texttt{WHERE}\ \{\ \llbracket s_1 \rrbracket_G\ \}$
$\llbracket \textbf{implies } s_1\ (\textbf{return } (\textbf{graphliteral } s_2)) \rrbracket = \texttt{CONSTRUCT}\ \{\ I\ \}\ \texttt{WHERE}\ \{\ \llbracket s \rrbracket_G\ \}$
$\quad$ given $\llbracket \texttt{DEFAULT} : s_2 \rrbracket_U = (I, \epsilon)$
$\llbracket \textbf{implies } s_1\ (\textbf{return } x) \rrbracket = \texttt{SELECT}\ x\ \texttt{WHERE}\ \{\ \llbracket s_1 \rrbracket_G\ \}$
$\llbracket \textbf{and } s_1\ s_2 \rrbracket = \llbracket s_1 \rrbracket\ ;\ \llbracket s_2 \rrbracket$

6. Evaluation of expressiveness by backward translation

We have defined in previous sections the translation from SQUALL to SPARQL, which demonstrate the adequacy of SQUALL for querying and updating RDF stores. However, it remains to evaluate the coverage of SPARQL expressiveness by SQUALL. We do so by sketching the backward translation from SPARQL to SQUALL. This demonstrates that SQUALL is as expressive as SPARQL 1.1 apart from arbitrary length paths and a few minor things (discussed at the end of this section). Note that the purpose of this backward translation is *not* to produce natural verbalizations of SPARQL queries, which is another issue [32], even though it could be used as a starting point.

SPARQL has a very rich syntax, but it can be simplified by transforming non-essential constructs to essential constructs. Turtle syntactic sugar is equivalent to basic graph patterns, and blank nodes can be replaced by variables. Property paths can also be expanded into graph patterns, except arbitrary length paths. In filters, usage of the function `IN` can be replaced by disjunction (`||`). A `VALUES`-clause can be moved into a graph pattern, and complex `VALUES`-patterns can be split into unions of joins of atomic `VALUES`-pattern (e.g., `VALUES ?x { :John }`). Expressions out of a graph pattern (e.g., `SELECT`, `ORDER BY`) can be moved into the graph pattern using `BIND` and a fresh variable. A `HAVING`-clause can be moved into the graph pattern as a filter, after reifying the aggregation as a subquery. References to graphs (e.g., `FROM`, `WITH`, `USING`) can be replaced by `GRAPH`-patterns to specify where triples should be queried or updated. Finally, in a `SELECT *` clause, the joker `*` can be replaced by all free variables in the graph pattern.

We present backward translation in a format similar to forward translation, but from SPARQL constructs to SQUALL phrases. SPARQL syntagms are typed with SQUALL

syntagms. The four forms of queries are translated to either interrogative or imperative sentences. Variables in the SELECT-clause are marked so that the determiner which is used upon their first occurence in the translation of the graph pattern. The solution modifier clause (e.g., ORDER BY DESC(?x) LIMIT 1) is translated into a adjective modifier to be used upon the first occurence of its variable (e.g., the highest thing ?x). The different forms of updates are translated to either declarative sentences, using a conditional for WHERE-clauses, or imperative sentences for graph-level updates.

$Query : Q$
    $\rightarrow$ ASK $Graph$ { whether $graph$? }
    $\rightarrow$ DESCRIBE URI { describe uri . }
    $\rightarrow$ DESCRIBE Var WHERE $Graph$ { describe var where $graph$. }
    $\rightarrow$ CONSTRUCT $Graph$ WHERE $Graph$ { return that $graph_1$ where $graph_2$. }
    $\rightarrow$ SELECT $Var \ldots Var$ WHERE $Graph$ $SolModif$ { $graph$? }

$Update : U$
    $\rightarrow$ INSERT DATA $Graph$ { $graph$. }
    $\rightarrow$ DELETE DATA $Graph$ { not $graph$. }
    $\rightarrow$ DELETE WHERE $Graph$ { if $graph$ then not $graph$. }
    $\rightarrow$ DELETE $Graph$ INSERT $Graph$ WHERE $Graph$ { if $graph_3$ then not $graph_1$ and $graph_2$. }
    $\rightarrow$ LOAD URI INTO GRAPH URI { load uri_1 into uri_2 . }
    $\rightarrow$ CLEAR GRAPH URI { clear uri . }
    $\rightarrow$ $Update$; $Update$ { $update_1$ $update_2$ }

Graph patterns are translated to sentences. Triple patterns translate to simple sentences, depending on whether a class or a property is used, and whether a variable or a URI is used. Classes are translated to nouns, properties to transitive verbs, and aggregations to aggregation adjectives. A variable ?x is translated as an apposition in a noun phrase (e.g., a thing ?x), on its first occurence, and then to a reference (e.g., ?x).

$Graph \;\; : S$
    $\rightarrow$ $Term$ a Var .   { $term$ belongs to var }
    $\rightarrow$ $Term$ a URI .   { $term$ is a uri }
    $\rightarrow$ $Term$ Var $Term$.   { var relates $term_1$ to $term_2$ }
    $\rightarrow$ $Term$ URI $Term$.   { $term_1$ uri $term_2$ }
    $\rightarrow$ BIND($Expr$ AS Var)   { var is $expr$ }
    $\rightarrow$ VALUES Var { $Term$ }   { var is $term$ }
    $\rightarrow$ $Graph$ $Graph$ { ($graph_1$ and $graph_2$) }
    $\rightarrow$ $Graph$ UNION $Graph$ { ($graph_1$ or $graph_2$) }
    $\rightarrow$ $Graph$ MINUS $Graph$ { ($graph_1$ and not $graph_2$) }
    $\rightarrow$ OPTIONAL $Graph$ { maybe $graph$ }
    $\rightarrow$ FILTER $Constr$ { $constr$ }
    $\rightarrow$ GRAPH $Term$ $Graph$ { at graph $term$ $graph$ }
    $\rightarrow$ SERVICE $Term$ $Graph$ { from service $term$ $graph$ }
    $\rightarrow$ { SELECT Var_1 ... Var_n ($Aggreg(Var_y)$ AS Var_x) WHERE $Graph$ }
        { var_x is the $aggreg$ things var_y such that $graph$ per var_1 and ... and per var_n }

Constraints are also translated to sentences, using the same SQUALL coordinations for Boolean operators as for relational algebra operators. Constraints simply produce

sentences based on predicates, where graph patterns produce sentences based on classes and properties. Therefore, unary predicates translate to nouns (like classes), and binary predicates translate to transitive verbs (like properties). Expressions translate to noun phrases, as they play the same role as variables and RDF terms.

$$Constr \; : \; S$$
$$\rightarrow Expr \; > \; Expr \; \{ \; expr_1 \; \text{is greater than} \; expr_2 \; \}$$
$$\rightarrow Pred1(Expr) \; \{ \; expr \; \text{is a} \; pred1 \; \}$$
$$\rightarrow Pred2(Expr,Expr) \; \{ \; expr_1 \; pred2 \; expr_2 \; \}$$
$$\rightarrow Constr \; || \; Constr \; \{ \; (constr_1 \; \text{or} \; constr_2) \; \}$$
$$\rightarrow Constr \; \&\& \; Constr \; \{ \; (constr_1 \; \text{and} \; constr_2) \; \}$$
$$\rightarrow \; ! \, Constr \; \{ \; \text{not} \; constr \; \}$$
$$\rightarrow \texttt{EXISTS} \; Graph \; \{ \; graph \; \}$$
$$\rightarrow \texttt{NOT EXISTS} \; Graph \; \{ \; \text{not} \; graph \; \}$$
$$Expr : NP$$
$$\rightarrow Expr \; \texttt{+} \; Expr \; \{ \; (expr_1 \; + \; expr_2) \; \}$$
$$\rightarrow Func(Expr, \; \ldots, \; Expr) \; \{ \; func(expr_1,\ldots,expr_n) \; \}$$
$$\rightarrow Literal \; \{ \; literal \; \}$$
$$\rightarrow Var \; \{ \; var \; \}$$

We have looked in detail at SPARQL's grammar[9] in order to identify all restrictions of SQUALL compared to SPARQL. The main restriction is arbitrary length property paths (e.g., (^author/author)+), which involves a form a recursivity that is difficult to express in natural language. Moreover, if recursivity could be expressed in SQUALL, it would probably exceed SPARQL's expressivity because recursivity only applies to property paths, and not to arbitrary graph patterns. The other restrictions of SQUALL are of lesser importance:

- n-ary DESCRIBE: e.g., DESCRIBE ?x ?y WHERE {?x a :man. ?x :spouse ?y.};

- n-ary ORDER BY: e.g., ORDER BY ASC(?age) DESC(?size);

- missing functions: &&, ||, !, UUID, STRUUID, COALESCE, IF, sameTerm;

- the third argument of REGEX is fixed to 'i';

- DISTINCT is always used with SELECT;

- SILENT is never used in updates and with SERVICE.

7. Evaluation on the QALD challenge

This section reports on the evaluation of SQUALL on the QALD-3 challenge, in which we took part. The QALD[10] challenge (Query Answering over Linked Data) provides "a benchmark for comparing different approaches and systems that mediate between a

---

user, expressing his or her information need in natural language, and semantic data".
The last campaign, QALD-3, provides hundreds of questions in natural language over
two large and real linked datasets: DBpedia [33] and MusicBrainz [34]. The principle
of the challenge is that a training set of 100 questions is provided, along with SPARQL
translations and answers, and systems are evaluated on a test set that is made of 100 new
questions. Systems are compared in terms of precision and recall for the test questions.

The QALD questions do not cover all features of SPARQL 1.1, and hence, do not
permit to evaluate all features of SQUALL. However, we find that the DBpedia questions
provide a rich set of questions in natural language on many topics, and cover a wide range
of SPARQL features including aggregations and solution modifiers. The missing features
are updates, named graphs, built-ins and expressions, graph literals (`CONSTRUCT` and
`DESCRIBE` queries), and collections.

The objective of our participation in the QALD-3 question answering task was to
evaluate the capability of SQUALL to express English questions in a natural and precise
way, and the precision and recall of its SPARQL translations. Because our approach relies
on a reformulation of the original questions, which is allowed by the QALD challenge,
the QALD measures of precision and recall are not enough to evaluate it. Therefore,
before presenting the performance of SQUALL in the QALD-3 challenge (Section 7.3),
we first assess the grammaticality of QALD questions in SQUALL (Section 7.1), and
the naturalness of their reformulation in SQUALL (Section 7.2). We do not evaluate the
efficiency of our system in detail as this is not an issue: the 100 questions of the challenge
are translated in 6s. We do not present either a usability study on the formulation of
SQUALL queries because we consider that guided input is necessary to overcome the
habitability problem [12], which is not the object of this paper and is already addressed
in existing work [13].

In our evaluation, we focus on the 99 English test questions (question 18 does not
exists in QALD-3) for the DBpedia dataset. Questions in other languages are out of the
scope of this paper. Training questions are very similar to test questions, and the same
conclusions can be drawn from them. For the MusicBrainz dataset, the definition of a
custom lexicon would be necessary to keep SQUALL questions natural because many
relations are reified as events.

7.1. Grammaticality of QALD questions

In this subsection, we assess the grammaticality of QALD questions in SQUALL. In
other words, we answer for each question whether it can be parsed as a SQUALL sentence
assuming the right lexicon and the right schema have been defined. For example, the first
question "Which German cities have more than 250000 inhabitants?" is a correct SQUALL
sentence, if we assume that "German" is an adjective, "cities" is a noun, and "inhabitants"
is a relational noun. It also assumes that there is a relation from each city to each
known inhabitant. This is unlikely in a dataset like DBpedia, but quite possible in other
contexts (e.g., in a civil registry). Therefore, grammaticality evaluates the coverage of
natural language by SQUALL, but not its adequation to a particular RDF dataset. It is
all about syntax, and none about lexicon.

Among the 99 test questions, we found that 57 of them are grammatical. Most
of the other 42 questions have only one ungrammatical element, and have therefore
a small edit distance to a grammatical question. We list below the different cases of
ungrammaticality, by decreasing frequency. In each case, we show an example where the

ungrammatical element is underlined, and add between brackets a possible replacement to make it grammatical.

- prepositions attached to nouns: "List all games <u>by</u> GMT." (relational adjective: made by);

- temporal and spatial phrases: "<u>When</u> did Michael jackson die?" (prepositional phrase: In which year);

- pronouns: "Which films starring Clint Eastwood did <u>he</u> direct <u>himself</u>?" (repeated proper noun: Clint Eastwood);

- proper noun as apposition: "Who created <u>the comic Captain America</u>?" (proper noun: Captain America);

- measure adjectives: "<u>How tall is</u> Michael Jordan?" (use relational noun: What is the height of);

- comparative adjectives: "<u>Was</u> the Cuban Missile Crisis <u>earlier</u> than the Bay of Pigs Invasion?" (use a relational noun: Did ... have an earlier date);

- composite nouns: "Give me the <u>Apollo 14 astronauts</u>." (noun group: astronauts with mission Appolo 14);

- void undefinite article: "Give me a list of all trumpet players that were <u>bandleaders</u>." (singular noun: a bandleader);

- perfect tenses: "How many space missions <u>have there been</u>?" (use past: were there);

The difficulty of those cases is less a syntactic problem than a semantic problem. For most cases, it would be relatively easy to extend SQUALL's grammar, but some of them involve implicit relations that can only be guessed from expertise on the domain and context. For example, "games by GMT" can mean "games produced by GMT" or "games developed by GMT".

7.2. Naturalness of SQUALL questions

The cost of using a CNL instead of spontaneous language is the need for reformulation, and the benefits is more precision and less ambiguity. We have reformulated the QALD questions into SQUALL sentences with the double objective to keep them as natural as possible, and to match the RDF schema of DBpedia. From the training phase, we had already learned some of the DBpedia vocabulary, and other URIs were found manually with Google searches and DBpedia browsing. We spent on average a few minutes per question for the reformulation phase. The SQUALL reformulations of the 99 test questions, as well as the reformulations of the 100 training questions, are available on SQUALL's homepage[11]. For illustration purposes, Table 4 lists a few original questions along with their SQUALL reformulation. From a detailed analysis of all reformulations, we derive the following conclusions.

---

[11]http://http://lisfs2008.irisa.fr/ocsigen/squall/examples: for each question, the SPARQL translation and answers from DBpedia can be obtained in two clicks.

1 "Which German cities have more than 250000 inhabitants?"
"Which Town that has country res:Germany has a populationTotal greater than 250000?"

2 "Who was the successor of John F. Kennedy?"
"Who was the successor of res:John_F._Kennedy?"

3 "Who is the mayor of Berlin?"
"Who is the leader of res:Berlin?"

4 "How many students does the Free University in Amsterdam have?"
"What is the numberOfStudents of res:Vrije_Universiteit?"

5 "What is the second highest mountain on Earth?"
"Which Mountain has the 2nd highest elevation?"

7 "When was Alberta admitted as province?"
"What is the dbp:admittancedate of res:Alberta?"

9 "Give me a list of all trumpet players that were bandleaders."
"Give me all Person-s whose instrument is res:Trumpet and whose occupation is res:Bandleader."

12 "Give me all world heritage sites designated within the past five years."
"Give me all WorldHeritageSite whose dbp:year is between 2008 and 2013."

15 "What is the longest river?"
"Which River has the highest dbp:length?"

21 "What is the capital of Canada?"
"What is the capital of res:Canada?"

26 "How many official languages are spoken on the Seychelles?"
"How many officialLanguage-s of res:Seychelles are there?"

28 "Give me all movies directed by Francis Ford Coppola."
"Give me all Film-s whose director is res:Francis_Ford_Coppola."

32 "How often did Nicole Kidman marry?"
"How many spouse-s of res:Nicole_Kidman are there?"

74 "When did Michael Jackson die?"
"What is the deathDate of res:Michael_Jackson?"

Table 4: A sample of original QALD-3 questions, followed by their SQUALL reformulation.

The concision of SQUALL is comparable to natural language. Table 5 compares the average length of questions in three languages: English (original QALD question), SQUALL (our reformulation of the questions), SPARQL (the golden standard provided by QALD organizers). Whereas SPARQL queries (excluding prologues) are two and a half times longer than natural language questions, SQUALL queries are only about 33% longer.

language	English	SQUALL	SPARQL
average question length	45	60	111

Table 5: Comparison of the average length of questions in the three languages.

The difference between natural language and SQUALL is in a large part explained by the namespaces in qualified names (e.g., `res:Berlin` instead of `Berlin`).

SQUALL queries look natural. The use of variables is hardly ever necessary in SQUALL (none was necessary in both training and test questions), while SPARQL queries are cluttered with many variables. No special notations were used, except for namespaces. Only grammatical words are used to provide syntax, and they are used like in natural language. There are 10 out of 99 questions where SQUALL is identical to natural language, up to proper names replaced by readable URIs, namespace prefixes, and plural marks (numbers are question ids): (2) "Who was the successor of res:John_F._Kennedy?", (14) "Give me all bandMember-s of res:Prodigy.", (21) "What is the capital of res:Canada?", (22) "Who is the dbp:governor of res:Wyoming?", (24) "Who was the dbp:father of res:Elizabeth_II?", (30) "What is the dbp:birthName of res:Angela_Merkel?", (54) "What are the dbp:nickname-s of res:San_Francisco?", (58) "What is the dbp:timezone of res:Salt_Lake_City?", (73) "How many child-s did res:Benjamin_Franklin have?", (76) "List the child-s of res:Margaret_Thatcher.".

Most discrepancies between natural language and SQUALL are a matter of vocabulary. Most discrepancies come from the fact that for each concept, a single word has been chosen in the DBpedia ontology, and related words are not available as URIs. Because our default lexicon uses URIs as nouns and verbs, some reformulation is necessary. In the simplest case, it is enough to replace a word by another: e.g., "wife" vs "dbp:spouse". In other cases, a verb has to be replaced by a noun, which requires changes in the syntactic structure: e.g., "Who developed Minecraft?" vs "Who is the developer of res:Minecraft?". An interesting example is "Who is the daughter of Bill Clinton married to?" vs "Who is the dbp:spouse of the child of res:Bill_Clinton?". The former question could be expressed in SQUALL if "marriedTo" was made an equivalent property to "dbp:spouse", and if "daughter" was made a subproperty of "child". In fact, this kind of discrepancy could be resolved, either by enriching the ontology with related words, or by preprocessing user sentences to replace spontaneous words by URIs. Alternately, the replacement could be done on the intermediate representation of even SPARQL query, with the benefit that the semantic role of words (e.g., class, property) would be explicit. The latter approach is analogue to and could reuse the techniques of ontology-based query answering systems (see Section 8).

Some discrepancies are deeper in that they exhibit conceptual differences between natural language and the ontology. We shortly discuss three cases:

- "List all episodes of the first season of the HBO television series The Sopranos!" vs "List all TelevisionEpisode-s whose series is res:The_Sopranos and whose seasonNumber is 1.". In natural language, an episode is linked to a season, which in turn is linked to a series. In DBpedia, an episode is linked to a series, on one hand, and to a season number, on the other hand. In DBpedia, a season is not an entity, but only an attribute of episodes.

34

- "Which caves have more than 3 entrances?" vs "Which Cave-s have an dbp:entranceCount greater than 3?". The natural question is a grammatical sentence in SQUALL, but it assumes that each cave is linked to each of its entrances. However, DBpedia only has a property "dbp:entranceCount" from a cave to its number of entrances.

- "Which classis does the Millepede belong to?" vs "What is the dbp:classis of res:Millipede?". The natural question is again a valid SQUALL sentence (after moving 'to' at the beginning), but it assumes that res:Millipede is an instance of a class that is itself an instance of dbp:classis. DBpedia does not define classes of classes, and therefore uses dbp:classis as a property from a species to its classis.

Those discrepancies are more difficult to solve. A first solution is to make the ontology better fit usage in natural language. A second solution is to apply transformations on the intermediate representation of natural SQUALL sentences so that it matches the ontology (e.g., transforming a count of entrances into the value of property dbp:entranceCount).

7.3. Performance in QALD-3 challenge

We here report on the results of our system SQUALL2SPARQL in QALD-3 challenge. They are published on the official website of the challenge. Other QALD-3 participants and their results are discussed in Section 8 (they are much lower but not directly comparable). We submitted the SPARQL translations produced by our system from our reformulations of the test questions to the QALD evaluation tool. Out of the 99 questions, we got the right answers for 80 questions (including the three OUT OF SCOPE questions), and partial answers for 13. Recall was 0.88, precision was 0.93, and the F-measure was 0.90. The errors come from:

- data heterogeneity (12 errors, questions 1, 6, 17, 19, 29, 33, 39, 60, 63, 72, 93, 96),

- the reformulation in SQUALL (2 errors, questions 14, 43),

- SQUALL2SPARQL (2 errors, questions 49, 59),

- the gold standard (2 errors, question 16, 75),

- the QALD endpoint (1 error, question 92).

Looking at heterogeneity errors in detail, it appears that most of them could be solved simply by: either adding generic super-properties in the DBpedia ontology, or by expanding common words (e.g., location, date) into UNION graph patterns. For example, in question 39 "Give me all companies in Munich.", the implicit relation "has location" can be translated in any of the three RDF properties: dbo:location, dbo:headquarter, dbo:locationCity. This explains why our reformulation in SQUALL "Give me all Company-es whose location is res:Munich." has recall 0.6 only (the default prefix was used for DBpedia ontology, so that location stands for dbo:location). If location, or another property, was defined as a super-property of the other properties, the same SQUALL question would have recall 1. Alternatively, assuming linguistic knowledge, the word "location" could be mapped to the graph pattern

```
{ ?x dbo:location ?y }
 UNION { ?x dbo:headquarter ?y }
 UNION { ?x dbo:locationCity ?y }
```

where `?x` and `?y` respectively stand for the subject and object of the relation. Such graph patterns could easily be exploited in the translation from the intermediate representation to SPARQL without the need to change the SQUALL language and its parsing.

Another problem related to heterogeneity is that some expected domain and range axioms are not verified in some cases. For example, in question 19 "Give me all people that were born in Vienna and died in Berlin.", 2 out of the 6 expected answers are not instances of the class `Person`. This is why our reformulation "Give me all Person-s whose birthPlace is res:Vienna and whose deathPlace is res:Berlin." missed 2 answers, even though it is arguably equivalent to the original formulation.

The errors coming from the reformulation of questions are due to the misspelling or misunderstanding of URIs. In question 14, res:Prodigy was used instead of res:The_Prodigy. In question 43, the property dbp:breed was used in the wrong direction.

The errors coming from SQUALL2SPARQL are due to an incorrect translation of the special verb "share". For example, Question 49 "Which other weapons did the designer of the Uzi develop?" was reformulated as "Which Weapon shares the dbp:designer with res:Uzi?", which returns Uzi itself as an answer. Another possible reformulation is "Which Weapon has the same dbp:designer as res:Uzi?", but it exhibits the same error.

The error from the endpoint is because the `BIND` construct of SPARQL is not (yet) supported by the QALD-3 endpoint. It is possible to write the SPARQL query to avoid it, but SQUALL2SPARQL relies on it to simplify the translation from SQUALL. Note that the correct answers are returned when using the official DBpedia endpoint.

8. Related work

In their evaluation of Natural Language Interfaces (NLI) and interfaces for the Semantic Web, Kaufmann and Bernstein [12] compare the usability of different approaches on a natural-formal scale (the *Formality Continuum*). This scale ranges from spontaneous natural language to formal query language, with Controlled Natural Language (CNL) in between. For formal languages (e.g., SPARQL) or CNLs, the lack of naturalness is compensated by assisting users in the formulation of queries (e.g., auto-completion). Interestingly, the participants of the QALD-3 challenge [14] cover a wide range of the formality continuum, even if most of them (4/6) fall in the "spontaneous natural language" category. We discuss each category in turn, starting with QALD-3 participants.

Spontaneous natural language. QALD-3 participants in this category are: CASIA [35] (F-measure = 0.36), RTV [36] (F-measure = 0.33), Intui2 [37] (F-measure = 0.32), and SWIP [38, 39] (F-measure = 0.17). Many other systems have been developed: e.g., PowerAqua [11], QAKiS [40], FREyA [10]. While they differ a lot in the details, they all share a similar architecture. First, a syntactic analysis is performed, usually based on standard NLP tools (e.g., Stanford NLP tools), in order to identify entities and relationships, and to generate a few triples. Second, a mapping from lexical forms to semantic forms is searched with the help of external resources, which are both linguistic (e.g., WordNet, Wikipedia, relational patterns), and ontologic (e.g., fixed ontology, Watson). Finally, one or several SPARQL queries are generated and ranked. In some cases, answers are directly produced and ranked without using a SPARQL query explicitly. Most of the effort is generally spent on the semantic mapping, and only a shallow syntactic parsing is performed. From QALD-3 results, CASIA and SWIP generate only 2-triples queries

(which cover 74 test questions), and RDF and Intui2 only 1-triple queries (60 questions). Only RTV supports counting (4 questions), and none of them supports comparatives (6 questions) or superlatives (6 questions). Among their wrong or missing results, roughly half can be accounted on syntax, and half on semantic mapping.

Controlled natural language. SQUALL is the only QALD-3 participant in this category (a former version of SWIP took part in QALD-1 as a CNL). It focuses on syntactic analysis, and SPARQL generation, and completely ignores (so far) semantic mapping. Therefore, it relies on a reformulation of original questions in the vocabulary of the target RDF dataset, and its score in the challenge (F-measure = 0.90) is hence not directly comparable to other participants' score. However, it demonstrates that complex natural questions can be reliably translated to SPARQL. SQUALL better handles complex syntactic constructs such as nested relative clauses, and coordinations. It can generate arbitrarily many triples, it supports all features of QALD questions (comparatives, superlatives, counting), and more (disjunction, negation, other aggregations, updates, etc.). Many CNLs have been defined in the past decades [17], but SQUALL is the only one that targets SPARQL queries and updates. Other CNLs for the Semantic Web rather target ontological axioms (facts and rules). ACE [19] is a general purpose CNL that can target various formalisms, and there are a number of more specialized CNLs targeting OWL axioms (e.g., SOS, Rabbit) [20]. In Kuhn's survey [17], SQUALL is evaluated as having: a high precision (P^5), a medium expressiveness compared to NL (E^3), natural sentences (N^4), and a short description (S^4). In comparison, a natural language is evaluated as $P^1E^5N^5S^1$, i.e., maximally expressive and natural, but minimally precise and short-defined; and SPARQL is evaluated as $P^5E^3N^1S^5$.

Guided navigation. Scalewelis [41] (F-measure = 0.33) is the only QALD-3 participant based on guided navigation. Like SQUALL, it is based on a reformulation of original queries, but users are guided step by step, and do not have to actually write the query. Guidance is based on actual data in the RDF store, and follows the principles of Query-based Faceted Search (QFS) [8], which allows for more expressiveness than other semantic faceted search systems (e.g., SlashFacet [6], BrowseRDF [7]). At each step, users are given a choice of relevant classes, properties, and entities to be inserted in the query under construction. Scalewelis' query language is a subset of SQUALL that supports nested relative clauses and conjunctive coordinations, but none of comparisons, superlatives, and aggregations. Errors in the QALD-3 challenge are explained by the lack of expressivity, and by the high heterogeneity of DBpedia's vocabulary. Ginseng [13], resp. GINO [42], guides users in the input of queries, resp. resource descriptions, through auto-completion of CNL sentences. However, their expressiveness is lower than SQUALL, and their guidance is based on an ontology rather than on actual data, which is less precise and requires a well-defined ontology.

9. Conclusion and perspectives

In the spectrum that goes from full natural language to formal languages like SPARQL, SQUALL (Semantic Query and Update High-Level Language) occupies a unique position. It offers the same expressiveness as SPARQL for querying and updating RDF data, and still qualifies as a controlled natural language (CNL). This means

that among the natural language interfaces, SQUALL is the one that is by far the most expressive; and that among the formal languages, SQUALL is the one that is the most natural. The current limit of SQUALL is that end-users have to comply with its controlled syntax, and have to know the RDF vocabulary (i.e., *Which are the classes and properties?*). However, the important result is that SQUALL can be used as a substitute for SPARQL because this entails no loss, neither in expressiveness, nor in precision.

Most existing systems try and solve the query answering challenge by addressing all aspects at once: syntactic analysis, mapping from lexical forms to semantic forms, contextual disambiguation, and SPARQL generation. We think that it may be more efficient, and that it would facilitate collaboration, to address those aspects separately. We have shown in this paper that syntactic analysis and SPARQL generation can reach the expressivity of SPARQL, while retaining a high level of naturalness (at the syntactic level). Future work will consist in integrating SQUALL with existing results about other aspects:

- *using domain-specific lexicons instead of the default one.* Such lexicons may be constructed manually, or generated automatically [43]. There is an RDF vocabulary, Lemon [44], to represent and share such lexicons.

- *applying mapping techniques on the intermediate representation.* SQUALL's intermediate representation is compatible with those of OQA systems, only being more complex combinations of triple patterns. In simple cases, mapping simply replaces words by URIs (e.g., "mayor" by `dbo:leader`). In more complex cases, mapping may replace combinations of triples by other combinations of triples (e.g., a count of "entrances" by the property `dbp:entranceCount`).

- *using SQUALL as an intermediate representation.* If full natural language is to be accepted as input, a solution is to use mature NLP tools (e.g., Stanford NLP parser) to parse a spontaneous sentence, and then translate the resulting dependency graph to SQUALL, which is arguably much closer to NL than SPARQL is.

- *using guided input and dialog.* Grammar-based guided input (auto-completion) ensures that only syntactically correct sentences are entered by users [13]. Another possible approach is to use query-based faceted search, which combines the guided exploration of faceted search and the expressivity of query languages [8]. If a lexicon is available, it can also ensures that only known words, and hence known concepts, are used. Finally, if data is available during guided input, it can be queried to rule out invalid interpretations, and the user can be asked to choose among the remaining interpretations. The combination of those mechanisms open the door for a rich dialog between the user and the system, but remains challenging in terms of complexity and scalability.

References

[1] S. Ferré, SQUALL: a controlled natural language as expressive as SPARQL 1.1, in: E. Métais (Ed.), Int. Conf. Application of Natural Language to Information Systems (NLDB), LNCS 7934, Springer, 2013, pp. 114–125.

[2] S. Ferré, SQUALL: a controlled natural language for querying and updating RDF graphs, in: T. Kuhn, N. Fuchs (Eds.), Controlled Natural Languages, LNCS 7427, Springer, 2012, pp. 11–25.

[3] P. Hitzler, M. Krötzsch, S. Rudolph, Foundations of Semantic Web Technologies, Chapman & Hall/CRC, 2009.

[4] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, in: I. F. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 30–43.

[5] SPARQL 1.1 query language, `http://www.w3.org/TR/sparql11-query/`, W3C Proposed Recommendation (2012).
URL `http://www.w3.org/TR/sparql11-query/`

[6] M. Hildebrand, J. van Ossenbruggen, L. Hardman, /facet: A browser for heterogeneous semantic web repositories, in: I. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 272–285.

[7] E. Oren, R. Delbru, S. Decker, Extending faceted navigation to RDF data, in: I. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 559–572.

[8] S. Ferré, A. Hermann, Reconciling faceted search and query languages for the Semantic Web, Int. J. Metadata, Semantics and Ontologies 7 (1) (2012) 37–54.

[9] V. Lopez, V. S. Uren, M. Sabou, E. Motta, Is question answering fit for the semantic web?: A survey, Semantic Web 2 (2) (2011) 125–155.

[10] D. Damljanovic, M. Agatonovic, H. Cunningham, Identification of the question focus: Combining syntactic analysis and ontology-based lookup through the user interaction, in: Language Resources and Evaluation Conference (LREC), ELRA, 2010.

[11] V. Lopez, M. Fernández, E. Motta, N. Stieler, PowerAqua: Supporting users in querying and exploring the semantic web, Semantic Web 3 (3) (2012) 249–265.

[12] E. Kaufmann, A. Bernstein, Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases, J. Web Semantics 8 (4) (2010) 377–393.

[13] A. Bernstein, E. Kaufmann, C. Kaiser, Querying the semantic web with Ginseng: A guided input natural language search engine, in: Work. Information Technology and Systems (WITS), 2005.

[14] P. Cimiano, V. Lopez, C. Unger, E. Cabrio, A.-C. N. Ngomo, S. Walter, Multilingual question answering over linked data (QALD-3): Lab overview, in: P. Forner, H. Müller, R. Paredes, P. Rosso, B. Stein (Eds.), Information Access Evaluation. Multilinguality, Multimodality, and Visualization - Int. Conf. CLEF Initiative, LNCS 8138, Springer, 2013, pp. 321–332.

[15] A. Hermann, S. Ferré, M. Ducassé, An interactive guidance process supporting consistent updates of RDFS graphs, in: A. ten Teije et al. (Ed.), Int. Conf. Knowledge Engineering and Knowledge Management (EKAW), LNAI 7603, Springer, 2012, pp. 185–199.

[16] N. E. Fuchs, R. Schwitter, Web-annotations for humans and machines, in: E. Franconi, M. Kifer, W. May (Eds.), European Semantic Web Conference, LNCS 4519, Springer, 2007, pp. 458–472.

[17] T. Kuhn, A survey and classification of controlled natural languages, Computational Linguistics.

[18] P. Smart, Controlled natural languages and the semantic web, Tech. rep., School of Electronics and Computer Science University of Southampton (2008).
URL `http://eprints.ecs.soton.ac.uk/15735/`

[19] N. E. Fuchs, K. Kaljurand, G. Schneider, Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces, in: G. Sutcliffe, R. Goebel (Eds.), FLAIRS Conference, AAAI Press, 2006, pp. 664–669.

[20] R. Schwitter, K. Kaljurand, A. Cregan, C. Dolbear, G. Hart, A comparison of three controlled natural languages for OWL 1.1, in: K. Clark, P. F. Patel-Schneider (Eds.), Workshop on OWL: Experiences and Directions (OWLED), Vol. 258, CEUR-WS, 2008.

[21] P. Haase, J. Broekstra, A. Eberhart, R. Volz, A comparison of RDF query languages, in: S. M. *et al.* (Ed.), Int. Semantic Web Conf., LNCS 3298, Springer, 2004, pp. 502–517.

[22] D. R. Dowty, R. E. Wall, S. Peters, Introduction to Montague Semantics, D. Reidel Publishing Company, 1981.

[23] R. Montague, Universal grammar, Theoria 36 (1970) 373–398.

[24] H. Barendregt, The Lambda Calculus, Its Syntax and Semantics, Vol. 103 of Studies in Logic and the Foundations of Mathematics, Elsevier, 1985.

[25] L. Damas, R. Milner, Principal type-schemes for functional programs, in: ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, ACM, 1982, pp. 207–212.

[26] D. Biber, S. Johansson, G. Leech, S. Conrad, E. Finegan, Longman grammar of spoken and written English, Pearson Education Limited, 1999.

[27] P.-Y. Hsu, D. S. P. Jr., Improving SQL with generalized quantifiers, in: P. S. Yu, A. L. P. Chen (Eds.), Int. Conf. Data Engineering, IEEE Computer Society, 1995, pp. 298–305.

[28] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge (MA), 1986.

[29] P. Wadler, Monads for functional programming, in: Advanced Functional Programming, Springer, 1995, pp. 24–52.

[30] H. Kamp, U. Reyle, From discourse to logic: Introduction to model theoretic semantics of natural language, formal logic and discourse representation theory, Kluwer, 1993.

[31] R. Muskens, Combining Montague semantics and discourse representation, Linguistics and philosophy 19 (1996) 143–186.

[32] A.-C. N. Ngomo, L. Bühmann, C. Unger, J. Lehmann, D. Gerber, Sorry, I don't speak SPARQL: translating SPARQL queries into natural language, in: WWW, 2013, pp. 977–988.

[33] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia - a crystallization point for the web of data, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3) (2009) 154–165.

[34] A. Swartz, Musicbrainz: A semantic web service, Intelligent Systems, IEEE 17 (1) (2002) 76–77.

[35] S. He, S. Liu, Y. Chen, G. Zhou, K. Liu, J. Zhao, CASIA@QALD-3: A question answering system over linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL http://www.clef2013.org

[36] C. Giannone, V. Bellomaria, R. Basili, A HMM-based approach to question answering against linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL http://www.clef2013.org

[37] C. Dima, Intui2: A prototype system for question answering over linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL http://www.clef2013.org

[38] C. Pradel, G. Peyet, O. Haemmerlé, N. Hernandez, SWIP at QALD-3: Results, criticisms and lesson learned, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL http://www.clef2013.org

[39] F. Amarger, O. Haemmerlé, N. Hernandez, C. Pradel, Taking SPARQL 1.1 extensions into account in the SWIP system, in: H. D. Pfeiffer, D. I. Ignatov, J. Poelmans, N. Gadiraju (Eds.), Int. Conf. Conceptual Structures, LNCS 7735, Springer, 2013, pp. 75–89.

[40] E. Cabrio, J. Cojan, A. P. Aprosio, B. Magnini, A. Lavelli, F. Gandon, QAKiS: an open domain QA system based on relational patterns, in: B. Glimm, D. Huynh (Eds.), Int. Semantic Web Conf. (Posters & Demos), Vol. 914 of CEUR Workshop Proceedings, 2012.

[41] J. Guyonvarch, S. Ferre, M. Ducassé, Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search, Research report PI-2009, IRISA (2013).
URL http://hal.inria.fr/hal-00868460

[42] A. Bernstein, E. Kaufmann, GINO - a guided input natural language ontology editor, in: I. F. C. et al. (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 144–157.

[43] S. Walter, C. Unger, P. Cimiano, A corpus-based approach for the induction of ontology lexica, in: Int. Conf. Applications of Natural Languages to Information Systems (NLDB), LNCS 7934, Springer, 2013, pp. 102–113.

[44] J. McCrae, D. Spohr, P. Cimiano, Linking lexical resources and ontologies on the semantic web with lemon, in: Extended Semantic Web Conference (ESWC), LNCS 6643, Springer, 2011, pp. 245–259.