Leveraging the Expressivity of Grounded Conjunctive Query Languages

Alissa Kaplunova, Ralf Möller, Michael Wessel

Hamburg University of Technology (TUHH)

Abstract. We present a pragmatic extension of a Semantic Web query language (including so-called grounded conjunctive queries) with a termination safe functional expression language. This addresses problems encountered in daily usage of Semantic Web query languages for which currently no standardized solutions exist, e.g., how to define aggregation operators and used-defined filter predicates. We claim that the solution is very flexible, since users can define and execute *ad hoc extensions* efficiently and safely on the Semantic Web reasoning server without having to devise and compile specialized "built-ins" and "plugins" in advance. We also address the scalability aspect by showing how aggregation operators can be realized efficiently in this framework.

1 Introduction

Nowadays, Description Logics (DLs) provide the basis for Semantic Web technology, and in particular, for the de facto standards for Semantic Web ontology languages such as OWL [1]. DL systems can thus be used as Semantic Web repositories. They offer a set of standard inference services, such as consistency checking, automatic computation of the concept (class) hierarchy (the so-called taxonomy), and the basic retrieval services (e.g., instance retrieval) [2]. An example of a prominent and widely-used DL system is the RACERPRO system [3] which implements the expressive DL $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$, also known as $\mathcal{SHIQ}(D^-)$. In the context of the Semantic Web, especially expressive semantic query languages (QLs) are of great importance to realize the vision of semantic information retrieval, which is at the heart of the Semantic Web idea. These QLs realize retrieval functionality that goes beyond the retrieval functionality offered by the basic retrieval services (e.g., instance retrieval).

Today, the most prominent Semantic Web QLs are (extended) RQL dialects, SWRL (the Semantic Web Rule Language) [4], SPARQL [5] and OWL-QL. RQL [6] is primarily an RDF QL and thus lacks many important expressive means and inference capabilities needed to query ontologies / documents written in more expressive Semantic Web languages, e.g. OWL. The same holds for SPARQL; many nowadays existing SPARQL implementations do not consider the inferred (axiomatic) triples in an RDFS document at all and from the standard it is not clear whether they should or "how complete" a SPARQL implementation w.r.t. the RDFS semantics should be. Only recently attempts are made to augment and enhance the expressivity and *inference-awareness* of SPARQL in such a way that it will become useful to query, e.g. OWL documents. This extension will be called SPARQL-DL. SWRL was primarily designed as a rule language, but can also be used as a query language. Full SWRL is undecidable, but restricted subsets (so-called DL-safe SWRL) exists. Considering the nowadays available SWRL implementations, the situation is similar as for SPARQL (e.g., sometimes, simple forward chaining rule engines are used to implement SWRL, so most of the inferences are missed). OWL-QL is very complex and does not seem to get much support from implementors, since its semantics is quite involved.

The native QL of the RACERPRO description logic system is called NRQL (new RACERPRO Query Language, [7]). It was primarily designed as a DL ABox query language and was later extended to also address specific aspects of OWL (despite its name, NRQL is not an extended RQL dialect). Due to its OWL capabilities, NRQL is also a Semantic Web QL. Being primary a QL for a DL system, NRQL was always fully inference-aware and thus provides expressive means not found in the QLs discussed above. Among others, nRQL offers classical and non-monotonic negation (so-called "negation as failure" or NAF-negation), extended concrete domain querying facilities, and a projection operator. Since NAF-negation and projection are available, both closed world and open world (universal and existential guarded) quantifications are available. Classical negation is not only applicable to (possibly complex) classes or concepts, but also to properties or roles. Classical and NAF-negation can even be mixed (e.g., one can retrieve those instances which are not known to be instances of $\neg mother$; these could be potential $mother s$).

Although NRQL is not standardized, it was and still is being used in many Semantic Web research projects and applications, not only because of its expressive means, but also due to its efficient and stable implementation which offers some unique features (queries are maintained as objects, multi-threading, etc.). However, in our own projects we found a need to extend the expressivity and practical relevance of NRQL even further. For example, *aggregation operators* were missing. Certain RQL dialects offer the standard SQL aggregation operators (`sum, max, min, count, avg,` etc.). Also for SWRL, there is the principle possibility to realize these through so-called "built-ins". For example, one could think of a specialized built-in atom such as

```
sum(?car,?weigth-of-parts,has-part;weight)
```

Given a binding for `?car`, the variable `?weigth-of-parts` is then bound to the sum of the `weight` datatype fillers of the `has-part` object property fillers of that `?car`. However, the semantics of such extensions is unclear (questions such as "which further predicates apply to `?weight-of-parts`?", "are variables typed?" etc. arise). Obviously, the wish list of such conceivable "extensions" is endless. The list of built-ins is therefore extensible in SWRL – user-defined built-ins can be added. The implementations of these atoms must be provided by users – a plugin architecture of the SWRL engine is thus required. The same idea applies of course to *filter* atoms or predicates (here, `filter(?x)` is true iff `?x` satisfies a certain user-defined `filter` predicate).

In order to offer a greater deal of flexibility and to allow for *ad hoc extensions*, we designed a more general extension mechanism for NRQL– (almost) arbitrary procedural extensions can be specified as part of a NRQL query. These extensions are written in a functional expression language called MINILISP. Even though MINILISP per se is purely functional we say *procedural* extension (see the Conclusion for further discussions).

The semantics of the MINILISP extension is defined in such a way that it does not interfere with the NRQL semantics. The nRQL query body is thus "kept clean". In the relational database realm, so-called stored procedures are well-known. However, stored procedures can result in *unsafe, non-terminating queries* (an unsafe query may run forever). Since decidability is crucial in the Semantic Web context, MINILISP is not a programming language, but a termination-safe functional expression language. MINILISP "programs" are executed efficiently on the RACERPRO server and thus offer the required efficiency and flexibility to "implement" arbitrary aggregation operators, filter predicates that cannot be formulated solely in the query language, create XML or HTML reports from query results, etc. In order to realize aggregation operators, often sub-queries have to be evaluated from within MINILISP programs. We introduce a new optimization technique (so-called promises) tailored for this purpose.

2 Theoretical Background

The class of conjunctive queries is well-known and established in the literature, e.g. see [9]. A *conjunctive query (CQ)* has the form $ans(\boldsymbol{X}) \leftarrow atom_1, \ldots, atom_n$, where $\boldsymbol{X} = (x_1, \ldots, x_m)$ is a variable vector. The expression $ans(\boldsymbol{X})$ is called the *head*, and $atom_1, \ldots, atom_n$ is the *body* of the query (interpreted as a conjunction). The $atom$s in the query body reference variables and/or individuals. All variables listed in $\boldsymbol{X}$ must also be mentioned in the body. The vector of body variables is denoted by $\boldsymbol{Y}$. Let $\boldsymbol{Z}$ denote those variables in $\boldsymbol{Y}$ which do not appear in $\boldsymbol{X}$.

Most Semantic Web QLs offer at least concept and role query atoms. If x and y are variables or individuals, then *concept query atoms* are unary atoms (e.g., $C(x)$), whereas *role query atoms* are binary atoms (e.g., $R(x, y)$). Often, also a (binary) *equality atom* is offered (either written as $x = y$, $= (x, y)$ or $same\_as(x, y)$).

A *query answer* is a set of (m-) tuples. Each tuple represents bindings for the head variables $\boldsymbol{X}$. In general, a head variable (in $\boldsymbol{X}$) is bound to an RDF node (representing an ABox individual), or an ABox individual. These individuals are denoted with $\mathsf{inds}(\mathcal{O})$ in the following. In order to compute the bindings for the variables in $\boldsymbol{X}$, all possible substitution functions $\alpha : \boldsymbol{X} \rightarrow \mathsf{inds}(\mathcal{O})^m$ are considered and applied to the query body. In case the resulting variable substituted query is entailed by the ontology, α denotes a result tuple. In *unrestricted conjunctive queries*, the variables in $\boldsymbol{Z}$ are bound to individuals in the interpretation domain $\Delta^{\mathcal{I}}$ of the logical models of the ontology. These variables are therefore considered as *existentially quantified*. In so-called *grounded (or restricted)* conjunctive queries (GCQs), the following simplification is made: Not only are the $\boldsymbol{X}$ variables bound to $\mathsf{inds}(\mathcal{O})$, but also the $\boldsymbol{Y}$ variables (and thus, all variables in that query).

The answer of a CQ can now be specified by the following simple set comprehension; please note that variables not mentioned in $\boldsymbol{X}$ and individuals ($\mathsf{inds}(\mathcal{O})$) remain unaltered by α (for such i, $\alpha(i) = i$ holds):

$$\begin{aligned} \{\, (i_1, \ldots, i_m) \mid \exists \alpha : \boldsymbol{X} \mapsto (i_1, \ldots, i_m), (i_1, \ldots, i_m) \in \mathsf{inds}(\mathcal{O})^m, \\ \mathcal{O} \models \exists \boldsymbol{Z}.\alpha(atom_1) \wedge \cdots \wedge \alpha(atom_n) \,\}. \end{aligned}$$

For grounded conjunctive queries, we simply change the domain of α from $\boldsymbol{X}$ to $\boldsymbol{Y}$ and remove "$\exists \boldsymbol{Z}$." from the first-order body formula. In grounded conjunctive queries, the standard semantics can be obtained for so-called *tree-shaped queries* by using corresponding existential restrictions in query atoms [8] (e.g., if y is not in $\boldsymbol{X}$, then the atom $R(x, y)$ can be replaced by $\exists R.\top(x)$; $\exists R.\top$ is a complex anonymous concept).

From a theoretical perspective, nRQL goes beyond grounded conjunctive queries. nRQL provides additional expressive means (see [7]), especially, NAF-negation, projection and union operators, concrete domain reasoning facilities (so-called *constraint checking atoms*), as well as a novel lambda-based expression language to be discussed in this paper.

Let us consider the following example query which will also explain some basics of lambda expressions and introduce the constraint query atoms as well. Suppose we want to retrieve the $woman$ instances which are at least 40 and which have children whose fathers are at least 8 years older than their mothers. Let us start with the query body

$$\begin{array}{l}(woman \sqcap \geq_{40} age)(x), has\_child(x,y), has\_father(y,f), has\_mother(y,m),\\ age(f, age_1), age(m, age_2), (\lambda(v_1,v_2) \bullet (v_2 + 8 \leq v_1))(age_1, age_2)\end{array}$$

Please note that the atom $(\lambda(v_1,v_2) \bullet (v_2 + 8 \leq v_1))(age_1, age_2)$ specifies an anonymous predicate with formal parameters v_1, v_2 which is applied to the actual arguments age_1, age_2. Unlike in SWRL or SPARQL, nRQL does not allow variables to be bound to anything else than ABox individuals in order to prevent semantic problems. Thus, the age_i cannot be used as variables. We therefore have to rewrite the body using a more complex lambda expression, utilizing $age(v_i)$ *terms* in the comparison predicate instead of simple v_i variables:

$$\begin{array}{l}(woman, \geq_{40} age)(x), has\_child(x,y), has\_father(y,f), has\_mother(y,m),\\ (\lambda(v_1,v_2) \bullet (age(v_2) + 8 \leq age(v_1)))(f,m)\end{array}$$

This translates more or less directly into concrete nRQL syntax:

```
(and (?x (and woman (min age 40))) (?x ?y has-child)
     (?y ?f has-father) (?y ?m has-mother)
     (?f ?m (constraint age age
                        (<= (+ age-2 8) age-1))))
```

Thus, a constraint query atom is very similar to a lambda expression. It is applied to ABox variables `?f, ?m`, whose actual values of the attribute `age` are then bound to the formal arguments in the constraint "lambda body", `age-1, age-2`. Concrete domain reasoning is used to check whether the concrete domain predicate holds. Note that there may be no concrete known values for the `age` attributes, or their values need not be unique, if only constraints are specified on them, e.g., only $age(betty) + 10 < age(charles)$ is known. This also explains why nRQL does not offer variables ranging over the concrete domain (their solutions resp. bindings could not be computed in all cases). Finally, a complete nRQL query (including a query head) $ans(x) \leftarrow \ldots$ is written as `(retrieve (?x) ...)`.

3 The Power of λ

Unfortunately, the number of predicates which can be constructed with `constraint` query atoms is quite limited, either in order to ensure decidability in the concrete domain reasoning engine in RacerPro, or simply because the required predicate is missing. Unfortunately, RacerPro does not offer user-defined concrete domains. Thus, it is even impossible to query for persons having a certain `firstname` (e.g., `"Betty"`), given that only the concrete domain attribute `fullname` exists. This is unfortunate, since in many cases full concrete domain reasoning is not required (i.e., if concrete datatype values are specified as "told values" in the ontology). This is where MINILISP comes into play.

The basic idea is simple: A nRQL query head may not only contain variables, but also *lambda applications*. Thus, a head is a vector $\boldsymbol{X} = (h_1, \ldots, h_m)$, where h_i is either a variable or a lambda application. Such a lambda application has the syntax $((\lambda(v_1, \ldots, v_p) \bullet \ldots)\; y_1, \ldots, y_p)$; the y_i are again variables, which also have to appear in the body of the query: $y_i \in \boldsymbol{Y}$. The answer of a GCQ with body $atom_1, \ldots, atom_n$, head $(h_1, \ldots, h_m)$ and $\boldsymbol{Y} = (x_1, \ldots, x_k)$ is then specified by the following set comprehension:

$$
\begin{aligned}
\{\, (j_1, \ldots, j_m) \mid\; & \exists \alpha : \boldsymbol{Y} \mapsto (i_1, \ldots, i_k), (i_1, \ldots, i_k) \in \mathsf{inds}(\mathcal{O})^k, \\
& \mathcal{O} \models \alpha(atom_1), \ldots \mathcal{O} \models \alpha(atom_n), \\
& \text{such that for all } l \in 1 \ldots m\text{:} \\
& \quad j_l = \alpha(h_l) \quad \text{if } h_l \text{ is a variable}, \\
& \quad j_l = ((\lambda(v_1, \ldots, v_p) \bullet \ldots)\; \alpha(y_1), \ldots, \alpha(y_p)) \\
& \quad \text{if } j_l = ((\lambda(v_1, \ldots, v_p) \bullet\; \ldots)\; y_1, \ldots, y_p) \\
& \qquad \text{and } j_l \neq \bot \,\}.
\end{aligned}
$$

So, instead of just returning the tuple $(i_1, \ldots, i_k)$, the h_l "functions" are applied and its results included in the constructed answer tuple at that position. This is very similar to the `mapcar` operation in COMMON LISP. In case h_k is a variable, its binding $\alpha(h_k)$ is included. Otherwise, h_k denotes a lambda application: $((\lambda(v_1, \ldots, v_p) \bullet \ldots)\; \alpha(y_1), \ldots, \alpha(y_p))$. Its result is included in the answer tuple at that position in case the lambda did not return $\bot$. Whenever $\bot$ is returned by the lambda, the whole answer tuple is rejected instead. Thus, lambdas can be used to specify arbitrary filter predicates. Answer sets consisting of unary tuples can also be considered as flat sets, and thus, the structure of the elements in the answer set can be defined completely by means of lambda expressions. Moreover, by posing sub-queries in lambda bodies, we can easily implement arbitrary aggregation operators, as demonstrated in Section 5.

Obviously, the expressivity of that extension depends on the admissible lambda bodies. It is well-known that an unrestricted use of lambda results in undecidability (e.g., consider the classical textbook example $((\lambda(x) \bullet (x\; x))(\lambda(x) \bullet (x\; x))$ which specifies an endless loop). Lambda applications are specified in MINILISP. In order to grant termination, lambdas itself are not first class citizens (which is the case in languages such as Scheme). We will now describe the flexibility and added value of MINILISP in concrete syntax by means of examples.

4 MINILISP by Example

Consider an ABox representing objects in a geographic information system having width and length, and we want to compute and return the area of these objects with a query. The individual `box1` has a width of 10 and a length of 20:

```
(define-concrete-domain-attribute width :type integer)
(define-concrete-domain-attribute length :type integer)
(instance box1 (and (equal width 10) (equal length 20)))
```

We can then query for the areas of the objects in this ABox as follows:

```
(retrieve (?x ((lambda (box)
                (let ((w (told-value-if-exists (width box)))
                      (l (told-value-if-exists (length box))))
                   (if (and w l) (* w l) :reject)))
                ?x))
    (?x (and (a width) (a length))))
```

The answer thus is: `(((?x box1) 200))`. The query body `(?x (and (a width) (a length)))` selects all ABox individuals which have – possibly only implicit – fillers of the concrete domain attributes `width` and `length`. However, only in case these fillers are "told" in the ontology (= syntactically explicit available) it is possible to also retrieve them. Their retrieval is then supported by means of functional expressions such as `(told-value-if-exists (width box))`. In case these attribute values are told, `w` and `l` are bound and `(* w l)` is computed and returned; otherwise, the `:reject` symbol is returned, so the result tuple is rejected (see $j_l \neq \perp$ in the set comprehension in Section 3). Of course, we could easily reject certain boxes (whose size is too big or too small). Thus, almost arbitrary ad hoc filter predicates can be specified.

We claim that MINILISP is easy to understand and use for readers which have some COMMON LISP experience. In a nutshell, MINILISP offers the following data types: numbers, symbols, strings and lists (and thus also trees). It supports conditional execution (`if, cond, when, unless, case`), structure mapping functions such as `maplist` (like `mapcar` in COMMON LISP, e.g., `(maplist (lambda (x) (1+ x)) '(1 2 3))` returns `(2 3 4)`), as well as standard functions borrowed from the host language COMMON LISP (arithmetic functions, list function, string processing functions, comparison and sorting functions, etc.). In order to grant termination, lambdas (as required for higher-order functions such as `maplist`) are not first class citizens (not data objects). Thus, `((lambda (x) (x x)) (lambda (x) (x x)))` simply gives a syntax error (`x` cannot be bound to the function object `(lambda (x) (x x))`). No unbounded loops can be specified (only mappings over finite structures). MINILISP is purely functional, although there is a notion of a state exploited in counting variables (there is no variable assignment, but `(incf count)` and `(decf count)`).

In principle, all RACERPRO API functions can be called from within a lambda body. This also applies to `retrieve` itself. Thus, nested queries can be posed. We will illustrate how aggregation operators can be implemented using nested queries.

5 Aggregation Operators in MINILISP

Consider the following example KB in which the compositional structure of a car is modeled, see Figure 1. A car has certain parts, and each part has a certain weight:

```
(define-primitive-role has-part :transitive t)
(define-concrete-domain-attribute weight :type real)

(instance mycar car)
(related mycar engine1 has-part)
(related engine1 cylinder-1-4 has-part)
(related mycar wheel-1-4 has-part)
```

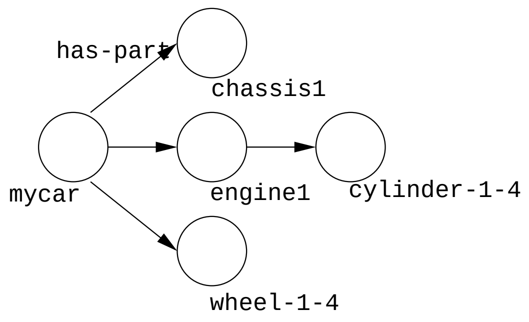

Fig. 1. Compositional structure of `mycar`

```
(related mycar chassis1 has-part)
(instance engine1 (= weight 200.0))
(instance chassis1 (= weight 400.0))
(instance wheel-1-4 (= weight 30.0))
```

Suppose we want to compute the overall weight of the car as well as the number of its components. Thus, using MINILISP, for each `?car` found we are going to construct two sub-queries. For a given `?car`, the first sub-query retrieves the components of that `?car` as well as their told weights, and the second sub-query simply counts the number of components. The two sub-queries are constructed and executed from within a MINILISP lambda body. Their results are then appropriately processed and returned.

Note that the bodies of the two sub-queries are almost identical for every `?car`, but the considered `?car` obviously changes. Thus, the bodies of the sub-queries are not fix. Sub-queries thus have to be constructed based on variable bindings which are established by outer (sub)queries. For this purpose, query templates can be constructed using the "backquote (`‘`) and comma (`,`) mechanism" from COMMON LISP. For example, if the variable `car` is bound to `mycar`, then the expression `‘(,car ?part hast-part)` evaluates to `(mycar ?part has-part)`. The query

```
(retrieve
  (((lambda (car)
     (let ((car-weight
            (reduce ’+ (flatten
                (retrieve ‘(((lambda (car-weight) car-weight)
                              (told-value-if-exists
                             (weight ?part))))
                         ‘(and (,car car) (,car ?part has-part)
                               (?part (a weight)))))))
           (car-parts (length
                (retrieve ‘(?part)
                           ‘(,car ?part has-part)))))
         ‘((?car ,car) (?no-of-parts ,car-parts)
           (?total-weight ,car-weight))))
        ?car))
     (?car car))
```

then returns

```
((((?car mycar) (?no-of-parts 4) (?total-weight 630.0)))).
```

The query works as follows. The body of the outer query consists of the single concept query atom (?car car). The lambda expression is then applied to the current binding of ?car. So, within the lambda body, car is bound to the binding of ?car. First, the total weight is computed. For this purpose, a sub-query is constructed. If ?car = mycar, then the query body '(and (mycar car) (mycar ?part has-part) (?part (a weight))) is constructed and posed, asking for the parts of mycar. The head of the sub-query consists of yet another lambda, which simply applies the told-value-if- exists head projection operator to retrieve the told values of the weight attribute of ?parts. The sub-query result is returned as a nested list; the list is flattened and its items are simply summed using (reduce '+ ...). We have computed the overall weight; this result is bound to the local variable car-weight. Similarly, the number of car-parts is computed (by posing yet another sub-query). Finally, the result of the lambda expression is constructed and returned. The constructed and returned value will become the result tuple. So, if car is mycar, and no-of-parts is 4, and car-weight is 630.0, then the template '((?car ,car) (?no-of-parts ,car-parts) (?total-weight ,car-weight)) constructs the result tuple (((?car mycar) (?no-of-parts 4) (?total-weight 630.0))). We can easily construct and return a string instead of a tuple by using (format nil "Car ~A has ~A parts and weights ~A kg." car no-of-pars car-weight). Thus, the answer is "mycar has 4 parts and weights 630.0 kg".

6 Efficient Aggregation Operators using Promises

Although the previous query demonstrated the power and utility of MINILISP, the aggregation was not computed efficiently, since for each binding of ?car, two new sub-queries were constructed. Thus, if 10 cars are present, 20 sub-queries had to be parsed, optimized, compiled and finally executed in order to compute the aggregation. NRQL supports the pre-compilation of queries; in fact, queries are maintained as first order objects which have a complex life cycle [7]. However, the two sub-queries cannot be simply precompiled since their bodies are not fixed. The bodies contain a variable part, ?car, whose binding can only be established at execution time by the outer query. From the perspective of the inner sub-queries, ?car is in fact an individual. Unfortunately, ?car cannot be treated as an individual at pre-compilation time, since the query compiler would then produce special code which will treat ?car as an individual, but after all, there is not individual ?car in the KB. This obviously prevents naive pre-compilation. Note that this situation does not arise in SQL engines.

A new optimization technique can help here which is a general technique that does not only apply to the NRQL engine. To the best of our knowledge, the technique has not been proposed or implemented before. A *promise* declares that certain variables have to be treated as individuals during query (pre-)compilation time. Thus, ?car can be treated as an individual by the optimizer and compiler. At the same time, it is *promised to* NRQL that this query will only be executed if a binding for ?car is established in advance, in this case, by the outer query. Thus, although the optimizer and compiler see and treat ?car as an individual, during execution time that individual can change (and so remains a variable). Thus, the query bodies are constant again, and only 2 bodies are needed instead of 20. Using promises, the example aggregation query looks as follows:

```
(with-future-bindings (?car)
    (prepare-abox-query (?part)
       (and (?car car) (?car ?part has-part))
       :id :parts-of-car-query)
    (prepare-abox-query
                  (((lambda (weight) weight)
                    (told-value-if-exists (weight ?part))))
       (and (?car car) (?car ?part has-part) (?part (a weight)))
       :id :weights-of-parts-of-car-query))
```

This prepares two queries named `:parts-of-car-query` and `:weights-of-parts-of-car-query`. The queries are compiled and optimized, but not executed yet. Since nRQL supports full life cycle management for queries, these queries are from now on available as query objects, ready for execution. Due to the surrounding lexical promise `with-future-bindings`, the query optimizer has treated the `?car` variable as an individual, although it must be handled as a variable at execution time. Thus, we have "promised" nRQL that we will only execute these queries if we supply a binding for `?car` in advance. We can establish such a binding during query execution using `with-nrql-settings` as follows:

```
(retrieve
   (((lambda (car)
        (with-nrql-settings (:bindings '((?car ,car)))
           (let ((car-weight
                  (reduce '+
                     (flatten
                       (execute-or-reexecute-query
                             :weights-of-parts-of-car-query)))))
                 (car-parts
                     (length
                          (execute-or-reexecute-query
                             :parts-of-car-query))))
             '((?car ,car) (?no-of-parts ,car-parts)
              (?total-weight ,car-weight))))
        ?car)))
 (?car car))
```

This results in a very efficient query execution, since (re)execution of a prepared query is immediate (only a function call to the compiled query evaluation function is required). No query parsing, optimization and compilation time is needed during query execution, and much less memory is used as well.

7 Conclusion

We have presented a pragmatic extension of a Semantic Web QL by lambda expressions. The termination safe functional expression language MiniLisp offers solutions to problems encountered in daily usage of Semantic Web QLs for which currently no standardized solutions exist. The proposed solution is technically sound, since the query body is kept clean from user defined predicates or procedural extension which might result in unsafe queries and semantical problems. The solution is flexible, since users can define and execute ad hoc extensions efficiently on the server without having to compile specialized "plugins" in advance. We have also addressed the scalability aspects by showing how aggregation operators can be realized efficiently in this framework by exploiting the novel notion of promises. Standard aggregation operators could also be made accessible as macros in this framework. We believe that the flexibility offered by MINILISP enhances the applicability of nRQL to real world problems. For example, the generation of HTML is directly supported in MiniLisp with "syntactic sugar".

Is MINILISP a *declarative* extension? The answer is *yes and no*. In case the specified lambda bodies are purely functional, the answer is truly yes. There are not states in MINILISP itself; for example, there is no variable assignment operator. Thus, typical MINILISP use cases will be fully declarative, e.g., if MINILISP is used for the realization of aggregation operators or filter predicates. However, for pragmatic reasons, MINILISP offers full access to all RACERPRO API functions. As such, it is of course possible to alter the state of a knowledge base while the query is still running. Since there is a notion of state involved, such queries can no longer be called fully declarative. One could argue that this kind of extension could also be executed on the client side. However, this will result in a bad performance and prevent ad hoc extensions. Moreover, optimization techniques such as promises cannot be used then.

8 Acknowledgments

We like to thank Atila Kaya for very valuable and thoughtful comments. This work was partially supported by the EU funded projects TONES (Thinking ONtologiES FET-FP6-6703) and BOEMIE (Bootstrapping Ontology Evolution with Multimedia Information Extraction, IST-FP6-027538).

References

1. van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference, http://www.w3.org/tr/owl-guide/ (2003)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Haarslev, V., Möller, R.: RACER System Description. In: Int. Joint Conference on Automated Reasoning, IJCAR '01. (2001)
4. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical Report, World Wide Web Consortium (2004)
5. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. Technical Report, World Wide Web Consortium (2006)
6. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: The Eleventh International World Wide Web Conference (WWW'02).
7. Wessel, M., Möller, R.: A High Performance Semantic Web Query Answering Engine. In: Proc. of the 2005 Description Logic Workshop (DL 2005)
8. Horrocks, I., Tessaris, S.: Querying the Semantic Web: a Formal Approach. In: Proc. of the 1st Int. Semantic Web Conf. (ISWC 2002).
9. Calvanese. D., De Giacomo, G., Lenzerini, M.: On the Decidability of Query Containment under Constraints. In: Prof. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98).