A High Performance Semantic Web Query Answering Engine

Michael Wessel and Ralf Möller
Hamburg University of Technology (TUHH)
Hamburg-Harburg, Germany
{mi.wessel | r.f.moeller}@tuhh.de

Abstract

We present an (extensively revised) semantic web query language called nRQL as well as a working high performance implementation of this language in the RacerPro system. We present the features of this query answering engine.

1 Introduction

The Racer description logic system [1] implements the very expressive description logic (DL) $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$. Racer's most prominent feature is the support for so-called *ABoxes* which allow for the representation of a "concrete state of the world" in terms of individuals and relationships. Due to Racer's ability to reason with ABoxes, the need for a more expressive *Abox querying language* became apparent. Many users of Racer have requested such a query language.

Thus, in dialog with the user community, we designed and implemented an expressive ABox query language tailored specifically for Racer in order to fulfill these requests. The result of our efforts is called the *n*ew *R*acer *Q*uery *L*anguage, nRQL (pronounce: nercle), which now is an integral part of the RacerPro 1.8.0 engine (the successor of Racer). nRQL was designed with a focus on conceptual simplicity as well as language orthogonality on the one hand, but also had to meet the requirement to offer "full query access" to all ABox features available in Racer, for example, offer access to the concrete domain part of an ABox.

Since its first release [2], nRQL has grown a lot. In this paper, we first present an extensively revised version of the nRQL language. However, the biggest achievement since [2] is the availability of a full-fledged nRQL query answering *engine*, offering an extensive API instead of just a single API query function "`retrieve`"as in [2]. The features of this new nRQL query answering engine are described as well in this paper. A benchmark of an older version

of the nRQL query answering engine using the *Lehigh University Benchmark (LUBM)* can be found in [3].
The *semantic web* is aimed at providing machine "understandable" meta data information for web resources. An other important aspect of Racer is its ability to process OWL (Web Ontology Language) documents (OWL KBs), which will be the official semantic web annotation language. Thus, Racer is not only a description logic system, but also a *semantic web reasoning engine*. Racer can process OWL Lite as well as OWL DL documents (knowledge bases) with approximations for nominals. Since *extensional information* from OWL documents (OWL instances and their interrelationships) are represented as ABoxes within Racer, it became apparent that nRQL can also be understood as a *semantic web query language*. In order to support special OWL features such as *annotation and datatype properties*, special OWL querying facilities have been incorporated into nRQL. Thus, nRQL can be used to query OWL documents.

The query language OWL-QL [4] is the W3C recommendation for querying OWL documents. nRQL has been used as a basic engine for implementing a subset of the OWL-QL query language [5]. We will describe the features of the nRQL engine which provide the basis for this work.

This paper is structured as follows. First, we informally describe the main features and language constructs of the revised nRQL language. We also motivate the design decisions we made. Then we specify the syntax and semantics of nRQL. After that, the features of the nRQL query answering engine will be presented. We then discuss related work and describe how OWL-QL can be supported by nRQL. Finally comes the conclusion and outlook. Throughout the remaining paper we assume basic knowledge of description logic terminology (e.g., we feel free to use notions like concept and role without defining them).

2 Informal Description of nRQL

Let us describe nRQL's main language features as well as the core design principles we have followed. We will use the KRSS-like native concrete nRQL / RacerPro syntax, since this will enable readers of this paper to perform hands-on experiments with nRQL immediately.

Basically, a nRQL query consists of a *query head* and a *query body*. For example, the query

```
(retrieve (?x ?y) (and (?x woman) (?x ?y has-child)))
```

has the head `(?x ?y)` and the body `(and (?x woman) (?x ?y has-child))`. It returns all mother-child pairs from the ABox which is queried.

In a nutshell, the nRQL *language* (to be distinguished from the nRQL *engine*, see below) can be characterized as follows:

1. *Variables and individuals* can be used in queries. The variables range over the individuals of an ABox (this is called the *active domain assumption*), and

must be bound against those ABox individuals which *satisfy* the query. The notion of satisfaction used in nRQL is defined in terms of logical entailment. This means, a variable will only be bound to an ABox individual iff it can be proven by Racer that this binding will hold in *all* models of the knowledge base. Returning to our example query body `(and (?x woman) (?x ?y has-child))`, `?x` will only be bound to those individuals which are known to be instances of the concept `mother` having a *known* child `?y` in *all* models of the KB.

nRQL distinguishes variables which must be bound to *differently named individuals* (e.g. `?x, ?y` cannot be bound to the same ABox individual) from variables for which this does not hold (e.g., `$?x, $?y`). Individuals from an ABox are identified by using them directly in the query, e.g. `betty`.

2. *Different types of query atoms* are available: these include concept query atoms, role query atoms, constraint query atoms, and same-as query atoms. To give some examples, the atom `(?x (and woman (some has-child female)))` is a concept query atom, `(?x ?y has-child)` is a role query atom, `(?x ?x (constraint (has-father age) (has-mother age) =))` is a constraint query atom (asking for the persons `?x` whose parents have equal age), and `(same-as ?x betty)` is a same-as query atom, enforcing the binding of `?x` to `betty`.

As the given example concept query atom demonstrates, it is possible to use compound concept expressions within concept query atoms. Regarding the role query atoms, the set of role expressions is more limited. However, it is possible to use inverted roles (e.g., role expressions such as `(inv R)`) as well as *negated roles* within role query atoms. Note that negated roles are not supported in the concept expression language of $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$; thus, they are only available in nRQL. For example, the atom `(?x ?y (not has-father))` will return those bindings for `?x, ?y` for which Racer *can prove* that the individual bound to `?x` cannot have the individual bound to `?y` as a father. If the role `has-father` was defined as having the concept `male` as a range, then at least all pairs of individuals in which `?y` is bound to a `female` person are returned, if `male` and `female` can be proven to be disjoint.

3. *Complex queries* are built from query atoms using the boolean constructors `and, union, neg`. We have already seen an example: `(and (?x woman) (?x ?y has-child))` is a simple *conjunctive query.* These constructors can be combined in an arbitrary way to create compound queries. This is why we call nRQL an orthogonal language.

The `neg` operator implements a *negation as failure semantics (NAF)*: `(neg (?x woman))` returns all ABox individuals for which Racer *cannot prove* that they are instances of `woman`. Thus, `(neg (?x woman))` returns the complement set of `(?x woman)` w.r.t. the set of all ABox individuals. If used in front of a role query atom, e.g. `(neg (?x ?y has-child))`, then this returns the two-dimensional set difference of `(?x ?y has-child)` w.r.t. the Cartesian product of all ABox individuals, i.e. all pairs of individuals which are not in the extension

of `has-child` in all models.

The semantics of nRQL ensures that DeMorgan's Laws hold:

`(neg (and` $a_1 \ldots\ a_n$`))` is equivalent to `(union (neg` a_1`)` ... `(neg` a_n`))`, and that the set difference using `neg` is well-defined for each basic query atom.

Please note that `(?x (not woman))` has a different semantics from `(neg (?x woman))`, since the former returns the individuals for which Racer *can prove* that they are *not instances* of woman, whereas the latter returns all instances for which Racer *cannot prover* that they are *instances* of woman. The same line of argumentation applies to role and constraint query atoms, although NAF negation of constraint query atoms is more involved in the presence of *role chains*.

4. *Support for retrieving* told concrete domain values *from the concrete domain part of a Racer ABox:* Suppose that `age` is a so-called *concrete domain attribute* of type integer. Thus, the `age` attribute fillers of a certain individual must be concrete domain values of type `integer`. We can use the following query to retrieve all adults as well as their ages: `(retrieve (?x (told-value (age ?x)) (?x (min age 18))))`, and a possible answer might be `(((?x michael) ((told-value (age michael)) 34)))`.

Since nRQL variables can only be bound to ABox individuals, but not to concrete domain datatype values, nRQL includes a special set of so-called *head projection operators* in order to support retrieval of concrete domain datatype values. These operators are denoted in a functional style in the head of a query - `(told-value (age ?x))` is such an operator. Moreover, a concrete domain value such as `34` can only be retrieved if it has been *told* to Racer - that is, the value must be explicitly (syntactically) specified in the ABox. The rationale for this is that Racer cannot compute the solutions of a concrete domain constraint system, but only decide its satisfiability. Even if Racer could do so, there would be no way to return these solutions as bindings to variables in the general case. Therefore, nRQL does not offer variables which range over concrete domain values. Allowing so would either destroy the orthogonality of nRQL, or make the language unsafe - after all, the semantics of queries such as `(retrieve (?x ?y) (and (?x ?y age) (?y (and integer (min 18)))))` is rather questionable. Note that concrete domain attributes are not roles; thus, `(?x ?y age)` is syntactically invalid. For these reasons, querying for concrete domain values is only supported by means of concept expressions and head projection operators.

Moreover, the constraint query atoms allow to "compare" the concrete domain attribute fillers of different individuals. Consider the query

```
(retrieve (?x (told-value (age ?x)))
      (and (?x (and woman (an age))) (?x ?y has-child)
           (?y ?y (constraint (has-father age) (has-mother age)
                              (< (+ age-1 8) age-2)))))
```

which returns the list of all woman and their ages, having children whose fathers

are at least 8 years older than their mothers. Note that `(has-father age)` denotes a "path expression": starting from the individual bound to `?y` we retrieve the value of the concrete domain attribute `age` of the individual which is the filler of the `has-father` role (feature) of this individual. In a similar way, the age of the mother of `?y` is retrieved. These concrete domain values are then used as actual arguments to evaluate the compound concrete domain predicate `(< (+ age-1 8) age-2)`. Here, `age-2` refers to `(has-mother age)`, and `age-1` to `(has-father age)`. Note that the suffixes `-1, -2` have been added to the `age` attribute in order to differentiate the two values. Obviously, this mechanism is not needed in case the two chains are ended by different attributes.

5. *Special support for querying OWL documents*, e.g., retrieving told datatype value fillers of OWL datatype and OWL annotation properties. Retrieval of these datatype values is supported in a similar style as in the concrete domain case, by means of concept query atoms and head projection operators.

Since concept expressions such as `(min age 18)` are useful in concept query atoms for specifying constraints on the datatype values to be retrieved, we have extended the Racer concept expression syntax to allow for similar concept query atoms containing datatype properties instead of attributes. This means, the concrete domain constraint expression language of Racer has been extended to also cover OWL datatype properties in addition to attributes. Thus, if `age` is not a concrete domain attribute, but now an OWL datatype property, then the previous query would work as well. However, querying for filler values of *annotation properties* is rather limited. The rationale is that, unlike datatype properties, *annotation properties* are not used for reasoning (the only serve documentary purposes). Thus, using concept query atoms, we can only query for the (non)existence of fillers of annotation properties of individuals. This means, if `AP` is an annotation property, then only concept query atoms such as `(?x (an AP))` and `(?x (no AP))` will work w.r.t. annotation properties. However, the filler annotation values of these individuals `?x` can then be retrieved by means of the `annotations` head projection operators as well (see syntax specification).

6. *nRQL also offers a body projection operator.* Sometimes this operator is required in order to reduce the "dimensionality" of a tuple set, for example, before computing a set difference with an n-dimensional set. This is exactly what happens if `neg` is used.

Let us motivate the necessity for such an operator: consider `(retrieve (?x) (and (?x mother) (?x ?y has-child)))`. This query returns all mothers having a *known child* in the ABox. Now, how can we query for mothers which do *not* have a *known* child?

Our first attempt will be the query `(retrieve (?x) (and (?x mother) (neg (?x ?y has-child))))`. A bit of thought and recalling that `(neg (?x ?y has-child))` returns the complement set of `(?x ?y has-child)` w.r.t. the Cartesian product of all ABox individuals will reveal that this query doesn't solve

the task. In a second attempt will would probably try `(retrieve (?x) (neg (and (?x mother) (?x ?y has-child))))`. However, due to DeMorgan's Law and nRQL's semantics, this query is equivalent to `(retrieve (?x) (union (and (neg (?x mother)) (?y top)) (neg (?x ?y has-child))))` – first the union of two two-dimensional tuple sets is constructed, and then only the projection to the first element of these pairs (`?x`) is returned. Obviously, this set contains also the instances which are *not* known to be mothers, what is wrong as well. Thus, the need for the projection operator becomes apparent: `(retrieve (?x) (and (?x mother) (neg (project-to (?x) (?x ?y has-child)))))` solves the task. This body projection operator was not present in earlier versions of nRQL, special syntax was introduced to address these problems, namely the unary special atoms `(?x (has-known-successor has-child))`, `(?x NIL has-child)` and `(NIL ?X child-of)`. These atoms (which still work) can now be seen as "syntactic sugar" for the bodies `(project-to (?x) (?x ?y has-child))`, `(neg (project-to (?x) (?x ?y has-child)))` and `(neg (project-to (?x) (?y ?x has-child)))`. The `project-to` operator can be used at any position in a query body.

7. *nRQL also offers defined queries.* These can be understood as a simple macro mechanism. For example, `(defquery mother (?x) (and (?x woman) (?x ?y has-child)))` can be used to define a query `mother` which can be subsequently used in calls such as `(retrieve (?x) (?x mother))` or in subsequent definitions, e.g. `(defquery married-mother (?x) (and (?x mother) (?x ?y has-spouse)))`. The definitions must be acyclic.

3 Syntax and Semantics of nRQL

Definition 1 (Syntax of nRQL) A nRQL query has a *head* and a *body*. A *head* can contain the following:

$$
\begin{array}{rcl}
\textit{head} & := & (\textit{head\_entry}^*) \\
\textit{object} & := & \textit{variable} \mid \textit{individual} \\
\textit{variable} & := & \text{a symbol beginning with ``?''} \\
\textit{individual} & := & \text{a symbol} \\
\textit{head\_entry} & := & \textit{object} \mid \textit{head\_projection\_operator} \\
\textit{head\_projection\_operator} & := & (\textit{cd\_attribute}\ \textit{object}) \mid \\
 & & (\texttt{told-value}\ (\textit{cd\_attribute}\ \textit{object})) \mid \\
 & & (\texttt{told-value}\ (\textit{datatype\_property}\ \textit{object}) \mid \\
 & & (\texttt{annotations}\ (\textit{annotation\_property}\ \textit{object}))
\end{array}
$$

The *body* is defined as follows:

$$
\begin{array}{rcl}
\textit{body} & := & \textit{atom} \mid (\ \{\texttt{and} \mid \texttt{union} \mid \texttt{neg} \mid \texttt{inv}\}\ \textit{body}^*) \mid \\
 & & (\texttt{project-to}\ (\textit{object}^*)\ \textit{body}) \\
\textit{atom} & := & (\textit{object}\ \textit{concept\_expr}) \mid (\textit{object}\ \textit{object}\ \textit{role\_expr}) \mid \\
 & & (\textit{object}\ \textit{object}\ (\texttt{constraint}\ \textit{chain}\ \textit{chain}\ \textit{constraint\_expr})) \mid
\end{array}
$$

(`same-as` *variable individual*)

chain := (*role_expr** *cd_attribute*)

The other constructs we defined in [2, 3] are expressible by the constructs we just gave (e.g., `(?x (has-known-successor R))` = `(project-to (?x) (?x ?y R))`, etc.) The "bridge" to the Racer syntax is given by the following rules:

concept_expr	:=	a Racer concept, with some extensions for OWL
role_expr	:=	a Racer role | (`inv` *role_expr*) | (`not` *role_expr*)
constraint_expr	:=	a (possibly compound) Racer concrete domain predicate, e.g. `(< (+ age-1 20) age-2)`
cd_attribute	:=	a Racer concrete domain attribute
datatype_property	:=	a Racer role used as OWL datatype property ■

Definition 2 (Semantics of nRQL) Let $\mathcal{A}$ be a Racer ABox, and $\mathcal{T}_\mathcal{A}$ denote its associated TBox. Denote the set of individuals used in $\mathcal{A}$ with $\mathsf{Inds}_\mathcal{A}$.

Let q be a nRQL query *body*. The function $\mathsf{vars}(q)$ is defined inductively: $\mathsf{vars}((x\ \ concept\_expr)) =_{def} \{x\}$, $\mathsf{vars}((x_1\ \ x_2\ \ role\_expr)) =_{def} \{x_1, x_2\}$, $\mathsf{vars}((x_1\ x_2\ (\texttt{constraint}\ \ldots))) =_{def} \{x_1, x_2\}$,
$\mathsf{vars}((\{\ \texttt{and}\ |\ \texttt{union}\ |\ \texttt{neg}\ |\ \texttt{inv}\ \}\ q_1 \ldots q_m)) =_{def} \bigcup_{1\leq i\leq m} \mathsf{vars}(q_i)$, **BUT** $\mathsf{vars}((\texttt{project-to}\ (x_1 \ldots x_m) \ldots)) =_{def} \{x_1 \ldots x_m\}$. Thus, vars "stops at projections". This is important.

Assume that $\langle x_{1,q}, \ldots, x_{n,q}\rangle$ is a lexicographic enumeration of $\mathsf{vars}(q)$. Denote the ith element in this vector with $x_{i,q}$, indicating its position in the vector.

Let $\mathcal{T}$ be a set of n-ary tuples $\langle t_1, \ldots, t_n\rangle$ and $\langle i_1, \ldots, i_m\rangle$ be an *index vector* with $1 \leq i_j \leq n$ for all $1 \leq j \leq m$. Then we denote the set $\mathcal{T}'$ of m-ary tuples with $\mathcal{T}' =_{def} \{\ \langle t_{i_1}, \ldots, t_{i_m}\rangle \mid \langle t_1, \ldots, t_n\rangle \in \mathcal{T}\ \} = \pi_{\langle i_1,\ldots,i_m\rangle}(\mathcal{T})$, called the *projection* of $\mathcal{T}$ to the components mentioned in the index vector $\langle i_1, \ldots, i_m\rangle$. For example, $\pi_{\langle 1,3\rangle}\{\langle 1,2,3\rangle, \langle 2,3,4\rangle\} = \{\langle 1,3\rangle, \langle 2,4\rangle\}$.

Let $\vec{b} = \langle b_1, \ldots, b_n\rangle$ be a *bit vector* of length n, $b_i \in \{0,1\}$. Let $m \leq n$. If $\vec{b}$ is a bit vector which contains exactly m ones, and $\mathcal{B}$ is a set, $\mathcal{T}$ is a set of m-ary tuples, then the n-dimensional *extension* $\mathcal{T}'$ of $\mathcal{T}$ w.r.t. $\mathcal{B}$ and $\vec{b}$ is defined as

$$\mathcal{T}' =_{def} \{\ \langle i_1, \ldots, i_n\rangle \mid \begin{array}{l}\langle j_1, \ldots, j_m\rangle \in \mathcal{T}, 1 \leq l \leq m, 1 \leq k \leq n \\ \text{with } i_k = j_l \text{ if } b_k = 1, \text{ and } b_k \text{ is the } l\text{th one (1) in } \vec{b}, \\ \text{otherwise, } i_k \in \mathcal{B}\ \ \}\end{array}$$

and denoted by $\chi_{\mathcal{B},\langle b_1,\ldots,b_n\rangle}(\mathcal{T})$. For example, $\chi_{\{a,b\},\langle 0,1,0,1\rangle}(\{\langle x,y\rangle\}) = \{\langle a,x,a,y\rangle, \langle a,x,b,y\rangle, \langle b,x,a,y\rangle, \langle b,x,b,y\rangle\}$. We denote an n-dimensional bit vector having ones at positions specified by the index set $\mathcal{I} \subseteq 1 \ldots n$ as $\vec{\mathbf{1}}_{n,\mathcal{I}}$. For example, $\vec{\mathbf{1}}_{4,\{1,3\}} = \langle 1,0,1,0\rangle$. Moreover, with $\mathcal{ID}_{n,\mathcal{B}}$ we denote the n-dimensional identity relation over the set $\mathcal{B}$: $\mathcal{ID}_{n,\mathcal{B}} =_{def} \{\ \underbrace{\langle x, \ldots, x\rangle}_{n} \mid x \in \mathcal{B}\ \}$.

The semantics of a nRQL query is given by the set of tuples it returns if posed to an ABox $\mathcal{A}$. This set of answer tuples is called the extension of q and

denoted by $q^{\mathcal{E}}$. We claim that the given semantics in terms of "tuple spaces" is substantially easier to grasp for users than logic-based semantics (i.e., the one we gave for the older versions of nRQL in [2, 3]).

In order to simplify the specification of the semantics, the query body q first undergoes some syntactical transformations: In a first step, q is rewritten by consistently replacing all its individuals with new representative fresh variables throughout the body. If the individual i has been replaced with the variable x_i, then we also add the conjunct (`same-as` x_i i) to q, yielding a body of the form (`and` q (`same-as` x_i i) (`same-as` ...) ...). In a second step, the *role chains* eventually present in constraint query atoms are decomposed. This means they are replaced by conjunctions of role query atoms such that only the concrete domain attributes remain in the chains of the constraint query atoms. Fresh anonymous variables are used for this purpose. E.g., the atom `(?x ?y (constraint (has-father age) (has-mother age) =))` is replaced with the body `(and (?x ?ano1-x-father has-father) (?y ?ano2-y-mother) (?ano1-x-father ?ano2-y-mother (constraint age age =)))`. Let q' be the transformed query.

We can now define the semantics of the different query atoms, being part of q'. Keep in mind that $\langle x_{1,q'}, \ldots, x_{n,q'} \rangle$ is the variable vector of q' and that n is the total number of variables returned by $\mathsf{vars}(q')$. For the sake of brevity we only consider variables which do not prevent the binding of an ABox individual to multiple variables (but the semantics can be easily extended). Thus, the semantics for the different nRQL query atoms is given as:

$$
\begin{aligned}
(q'_{x_i}\ \mathit{concept\_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i\}}}(\mathsf{concept\_instances}(\mathcal{A}, \mathit{concept\_expr})) \\
(q'_{x_i}\ q'_{x_j}\ \mathit{rolen\_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i,j\}}}(\mathsf{role\_pairs}(\mathcal{A}, \mathit{role\_expr})),\ \text{if } i \neq j \\
(q'_{x_i}\ q'_{x_i}\ \mathit{role\_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i\}}}(\mathsf{role\_pairs}(\mathcal{A}, \mathit{role\_expr}) \cap \mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}}) \\
(\texttt{same-as}\ q'_{x_i}\ \mathit{ind})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i\}}}(\{\mathit{ind}\}) \\
(q'_{x_i}\ q'_{x_j}\ (\texttt{constraint}\ \mathit{attrib}_1\ \mathit{attrib}_2\ P))^{\mathcal{E}} &=_{def} \\
&\quad \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i,j\}}}(\mathsf{predicate\_pairs}(\mathcal{A}, \mathit{attrib}_1, \mathit{attrib}_2, P)),\ \text{if } i \neq j \\
(q'_{x_i}\ q'_{x_i}\ (\texttt{constraint}\ \mathit{attrib}_1\ \mathit{attrib}_2\ P))^{\mathcal{E}} &=_{def} \\
&\quad \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{i}}_{n,\{i\}}}(\mathsf{predicate\_pairs}(\mathcal{A}, \mathit{attrib}_1, \mathit{attrib}_2, P) \cap \mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}})
\end{aligned}
$$

This definition uses some auxiliary functions, providing the bridge to the classical ABox retrieval functions offered by Racer. These bridge functions have the standard DL semantics in terms of logical entailment. However, as already mentioned, a nRQL *role expression (role_expr)* can also be a *negated* (or inverse) $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ role. A *predicate* (P) can be denoted as a lambda expression and is made from the vocabulary which Racer offers for building linear (in)equalities. For example, `(< (+ 20 age-1) age-2)` can be expressed as $P = \lambda(age_1, age_2).age_1 + 20 < age_2$:

$$
\begin{aligned}
&\mathsf{concept\_instances}(\mathcal{A}, \mathit{concept\_expr}) =_{def} \\
&\quad \{\, i \mid\ i \in \mathsf{Inds}_{\mathcal{A}}, (\mathcal{A}, \mathcal{T}_{\mathcal{A}}) \models \mathit{concept\_expr}(i) \,\}
\end{aligned}
$$

$\mathsf{role\_pairs}(\mathcal{A}, role\_expr) =_{def}$
$\quad \{ \langle i,j \rangle \mid i,j \in \mathsf{Inds}_\mathcal{A}, (\mathcal{A},\mathcal{T}_\mathcal{A}) \models role\_expr(i,j) \}$
$\mathsf{predicate\_pairs}(\mathcal{A}, attrib_1, attrib_2, P) =_{def}$
$\quad \{ \langle i,j \rangle \mid i,j \in \mathsf{Inds}_\mathcal{A}, (\mathcal{A},\mathcal{T}_\mathcal{A}) \models$
$\qquad \exists x,y : attrib_1(i,x) \wedge attrib_2(j,y) \wedge P(x,y) \}$

The semantics of compound nRQL bodies can be defined easily now:

$$\begin{aligned}
(\texttt{and}\ q_1 \ldots q_i)^\mathcal{E} &=_{def} \bigcap_{1\le j\le i} q_j^\mathcal{E} \\
(\texttt{union}\ q_1 \ldots q_i)^\mathcal{E} &=_{def} \bigcup_{1\le j\le i} q_j^\mathcal{E} \\
(\texttt{neg}\ q)^\mathcal{E} &=_{def} (\mathsf{Inds}_\mathcal{A})^n \setminus q^\mathcal{E} \\
(\texttt{inv}\ q)^\mathcal{E} &=_{def} \mathsf{inv}(q^\mathcal{E}),\ \text{where inv reverses all tuples} \\
(\texttt{project-to}\ (x_{i_1,q} \ldots x_{i_k,q})\ q)^\mathcal{E} &=_{def} \pi_{\langle i_1,\ldots,i_k\rangle}(q^\mathcal{E})
\end{aligned}$$

So far we have specified the semantics of a query body. To get the answer of a query, the *head* has to be considered. This can be seen as a further projection of $q^\mathcal{E}$ to the variables mentioned in the head. If the head contained an individual, then this individual has also been replaced by the representative variable in the head (see above). In case a head projection operator is encountered, the functional operator is applied to the binding of the argument variable. The value is included in the query answer (producing nested binding lists). In case of OWL datatype fillers, the projection operators will eventually return a list of datatype values. A formal definition of this process is omitted here due to space limitations. ■

4 The nRQL Engine

The nRQL query answering engine implements the nRQL language as part of the *RacerPro 1.8.0* system. Let us describe the features of this engine:

1. The engine offers *different querying modes:* basically, a synchronous *set-at-a-time mode* and an asynchronous *tuple-at-a-time mode.* In the *set-at-a-time mode*, a call to a querying function such as `retrieve` works synchronously. The client has to wait, the whole answer set is delivered in a bunch. However, many client applications prefer an asynchronous API: the *tuple-at-a-time mode* allows for incrementally loading the answer *tuple by tuple.* Thus, a call to `retrieve` will return immediately with a so-called *query identifier.* This identifier, say `:query-123`, can then be used as argument to functions such as `(get-next-tuple :query-123)`. Functions like `(get-next-n-tuples 10 :query-123)` are also provided. A similar way of client-server interaction is also presupposed in the OWL-QL query answering dialog. Thus, nRQL supports the OWL-QL query answering dialog quite well in this respect.

The incremental tuple-at-a-time mode can be used either *lazy or eager*: in the lazy mode, the next tuples will not be computed before requested by the client, unlike the eager mode, which pre-computes the next tuple(s) and puts them into a queue for future requests. In principle, there can be more than

one running query at a time. The nRQL engine allows for concurrent querying. When a query is executed, a *thread* from a *pool of threads* is acquired and put to work. The engine can process up to a few hundred queries simultaneously, and serializes and minimizes the calls to the basic Racer ABox retrieval functions (e.g., `concept-instances`), by using locking techniques and dedicated index structures. Nearly all answers from Racer are cached, but the index structures are automatically invalidated if changes to the ABox (or TBox) occur. If a KB changes while queries are still active, then nRQL can be advised to deliver a *KB-has-changed-warning token* to the clients.

2. The *degree of completeness of query answering in nRQL is configurable:* If ABoxes get very big, it becomes impossible to use Racers ABox retrieval functions for query answering. Even the required ABox consistency check will fail. Thus, nRQL can no longer be used in its complete mode. The incomplete modes can help, if "complete enough" for the application. These modes can still achieve a great deal of completeness. For example, using the incomplete nRQL mode "2", we observed in [3] that nRQL is still more complete than "DLDB" on the LUBM and achieves a comparable performance, even for "ABoxes" with a few 100.000 individuals.

Having incomplete modes available gives nRQL the ability to distinguish between cheap and expensive tuples. It is possible to advise nRQL first to deliver a set of *cheap tuples*, yielding an incomplete answer ("phase one"), and then to turn on Racers expensive ABox retrieval functions to deliver the remaining *expensive tuples* ("phase two"). We talk of *two-phase query processing modes.* Again, nRQL can be advised to deliver a *warning token* before phase two starts, informing the client that computation of the remaining tuples will eventually be expensive. The client can then chose to retrieve these additional tuples or not.

3. The engine supports *full life-cycle management for queries:* queries can be prepared, made active, be suspended or aborted, eventually terminate, can be resurrected, etc. Runtime resources are configurable (size of thread pool, maximum bound on the number of answer tuples setable, timeout setable, tuple permutations can be excluded, etc.).

4. Another important feature of the engine is the *built-in query optimizer.* The basic idea is to reorder the query atoms in a conjunctive query in such a way as to heuristically minimize the number of *generators* required to compute candidate bindings for the variables. For example, for the query `(and (?x ?y R) (?y D) (?x C))`, the execution plan `(?x c), (?x ?y R), (?y D)` is preferable over the plan `(?x C), (?y D), (?x ?y R)`, since the second plan has to compute a Cartesian product, whereas the first plan can, once a binding for `?x` has been established, simply enumerate the `R` successors of `?x` for `?y` candidate generation, which is much more "local". In order to decide whether `(?y D), (?y ?x (inv R)), (?x C)` might even be better, nRQL uses ABox statistics and information from previously evaluated queries in order to implement the "most

constrained generator first" heuristics.

5. nRQL can be advised to maintain a so-called *query repository.* This DAG records *query subsumption* relationships. nRQL will exploit cached answer sets of previously answered subsuming and subsumed queries. Reasoning with queries (e.g., consistency checking and computation of query subsumption) is incomplete in nRQL, but does not effect the usefulness of this cache.

6. The nRQL engine also offers so-called *complex TBox queries,* as well as a *simple rule mechanism.* However, no rule application strategy is presupposed. The antecedent of a rule is basically a nRQL query, and the consequence consists of a set of generalized ABox assertions.

7. nRQL also offers a set of so-called *data substrate representation layers.* Associating an ABox with such a layer enables users to associate "data values" with ABox individuals and role relationships, similar to OWL annotation properties. For example, it becomes possible to ask for the annotation values from an OWL document which contain the substring "name". Such a query is impossible on the ABox side. nRQL supports so-called *hybrid queries* which can be used to retrieve and combine information from the ABox with information from the corresponding substrate layer [6].

5 Related Work & Conclusion

nRQL has been influenced by the query language of LOOM [7]. However, unlike the LOOM query language, nRQL is specified in a formally rigorous way.

Basic conjunctive query languages for less expressive description logics than $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ (or OWL) have been been formally investigated in [8]. There, so-called distinguished as well as non-distinguished variables were introduced. The distinguished variables correspond to nRQL's variables. The non-distinguished variables are purely existential variables. For example, if `!y` denotes such a non-distinguished variable, then the query `(and (?x c) (?x !y R))` is basically equivalent to `(?x (and c (some r top)))`. This observation also suggests a technique for "removing" non-distinguished variables, the so-called "rolling up" technique. However, this procedure only works for acyclic queries, and becomes more technically involved if more than one non-distinguished variable is present in the query. These theoretical techniques have been used as a basis for the implementation of an experimental DQL (DAML Query Language) server [9]. DQL is the predecessor of OWL-QL, and quite similar in many respects. However, this server can only handle acyclic conjunctive queries.

OWL-QL [4], like DQL, offers *must-bind (= distinguished), may-bind, and do-not-bind (= non-distinguished) variables.* An OWL QL query contains a so-called *query pattern* which is a collection of OWL statements (a conjunction) in which some URI references or literals have been replaced by variables. May-bind variables are a "mixture" of must- and do-not-bind variables and are expected to return bindings if satisfying instances are known (otherwise they are treated

as do-not-binds). Since OWL statements not only represent ABox assertions, but also axioms analog to TBox axioms, this implies that OWL-QL queries can also be used to retrieve the subconcepts of a concept (so-called "structural queries"), etc. Variables therefore not necessarily range over OWL instances (ABox individuals). An OWL-QL implementation which uses nRQL as a basis could exploit the complex nRQL TBox queries in order to answer some of these structural queries. In principle, we believe that the semantic differences of must-bind, do-not-bind and may-bind variables will be rather hard to comprehend for end users of such systems. The lack of do-not-bind variables in nRQL can be compensated for in most cases by using existential role restrictions concepts in concept query atoms. An automatic "rolling-up procedure" is not yet implemented in nRQL.

References

[1] Haarslev, V., Möller, R.: RACER System Description. (In: Int. Joint Conference on Automated Reasoning, IJCAR '01)

[2] Haarslev, V., Möller, R., Straeten, R.V.D., Wessel, M.: Extended Query Facilities for Racer and an Application to Software-Engineering Problems. (In: Proc. of the Int. Workshop on Description Logics, DL '04)

[3] Haarslev, V., Möller, R., Wessel, M.: Querying the Semantic Web with Racer + nRQL. (In: Proc. of the KI-04 Workshop on Applications of Description Logics 2004, ADL '04)

[4] Fikes, R., Hayes, P., Horrocks, I.: OWL-QL - A Language for Deductive Query Answering on the Semantic Web. Technical Report KSL-03-14, (Knowledge Systems Lab, Stanford University, CA, USA)

[5] Galinski, J., Kaya, A., Möller, R.: Development of a Server to Support the Formal Semantic Web Query Language OWL-QL. (In: Proc. of the Int. Workshop on Description Logics, DL '05)

[6] Wessel, M.: Some Practical Issues in Building a Hybrid Deductive Geographic Information System with a DL Component. (In: Proc. of the 10th Int. Workshop on Knowledge Representation meets Databases, KRDB '03)

[7] MacGregor, R., Brill, D.: Recognition Algorithms for the Loom Classfier. (In: Proc. of AAAI'92, Thenth Conference on Artificial Intelligence)

[8] Horrocks, I., Tessaris, S.: Querying the Semantic Web: a Formal Approach. (In: Proc. of the 13th Int. Semantic Web Conf. ISWC '02)

[9] Glimm, B., Horrocks, I.: Query Answering Systems in the Semantic Web. (In: Proc. of the KI-04 Workshop on Applications of Description Logics 2004, ADL '04)