A Query Language for a Metadata Framework about Mathematical Resources

Ferruccio Guidi and Irene Schena

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
{fguidi,schena}@cs.unibo.it

Abstract. Our aim is in the spirit of the Semantic Web: to generate an on-line database of machine-understandable mathematical documents, re-using the same methodology and standard technologies, as RDF, mostly based on the exploitation of metadata to improve smart searching and retrieving. In this paper we describe the syntax and semantics of a query language which gives access to a search engine capable of retrieving resources on the basis of meta-information: the role of metadata consists in filtering the relevant data.

1 Introduction

The mathematical information, owing to its semantically rich *structured* nature, its scalability and inter-operability issues, can be considered as a relevant and interesting test case for the study of content-based systems and the development of the Semantic Web.
The Semantic Web[1] is an extension of the Web in which information is given a well-defined meaning in a machine-processable form, say by means of *metadata*, in order to share, interpret and manipulate it world-wide, enabling and facilitating specific functionalities such as searching and indexing.

Our aim is to generate an on-line database of machine-understandable mathematical documents, re-using the same methodology and standard technologies, as RDF, mostly based on the exploitation of metadata to improve smart searching and retrieving.

Since we are developing our work inside the HELM[2] project [1], which is now integrated with related projects in the framework of the MOWGLI Project[3], we have pointed out and taken into account different kinds of involved domains of application: general mathematical applications, as reviewing databases or on-line libraries, automated mathematics applications, as proof-assistants or proof-checking systems, and educational applications, as learning environments. All these applications require classifying, smart searching and browsing semantically meaningful mathematical information.

[1] <http://www.w3.org/2001/sw/>.
[2] Hypertextual Electronic Library of Mathematics, <http://helm.cs.unibo.it/>.
[3] Math On the Web: Get it by Logic and Interfaces, European FET Project IST-2001-33562, <http://mowgli.cs.unibo.it/>.

At this point we have to face two fundamental related issues: the meta-description[4] associated to data and an outfit query language.

We have distinguished at least three different levels of querying mathematical information:

– Querying metadata: filtering relevant data (see MathQL Level 1 in Sect. 2).
– Querying data: taking into account the semantically meaningful content of mathematics (i.e. pattern matching).
– Querying data “up to”: processing data on the basis of the *formal* content of mathematics (i.e. isomorphisms, unification, δ-expansion).

The standard format for metadata proposed by W3C is RDF (Resource Description Framework) [4,5]. RDF provides a common model for describing the semantical information on the Web corresponding to different domains of application: RDF allows anyone to express assertions (i.e. statements as subject-predicate-object records) about every resource on the Web.

In this paper we focus on a query language that gives access to a search engine capable of retrieving resources on the basis of meta-information, successfully querying RDF metadata eventually coming from different domains of application.
In particular, besides the main requirements for an RDF query language [10,3, 6], our query language on metadata fulfills two main design goals:

– abstracting from the representation *format* of the queried (meta)data (i.e. XML [9])
– abstracting from the concrete specification of *RDF* but supporting its formal model and characteristics, that are independence, interchange and scalability. Reasoning cannot be based on RDF, but we want an abstract logical model for our query language: this model is given in terms of relationships between resources (see 2).

2 MathQL Level 1

In this section we describe the features of MathQL-1 giving its syntax and semantics. Here are the main guidelines concerning the general language architecture.

– A resource is denoted by a URI reference [7] (or just reference), that is a URI (Uniform Resource Identifier) [7] with an optional fragment identifier, and may have some named attributes describing it. These attributes hold multiple string values and are grouped into sets containing the ones that are meaningful in the same context.
– A query gives a set of distinct resources each with its sets of attributes.
– The basic RDF data model is based on the concept of property: in an RDF statement (s, p, o) the property p plays the central role of the predicate that has a subject s and an object o. This model draws on well-established principles from various data representation communities. An RDF property can be

[4] For a detailed description of metadata and RDF schemas for HELM see [6].

thought as an attribute of a resource, corresponding to traditional attribute-value pairs. At the same time, a property can represent relationships between resources, as in an entity-relationship model. In this sense our language is correct and complete with respect to querying an abstract RDF data model. The language allows to build the set of attributed references involved by a property-relationship. In particular we regard a relation between values (that can be resources) and resources as a named set of triples (v, r, B) where v is a string value, r is a reference and B is a set of attributes (see 2.1), providing additional information on the specific connection between v and r (this is the case of a structured property or a structured value of a property).
Moreover the language allows to access any property of a resource by means of functions that we regard as named sets of couples (r, v) where r is a reference and $\{v \mid (r, v)\}$ is its associated multiple string value.
Examples of possible relations and functions can be found in section 3.1 where we discuss an implementation of the language.
– Queries and query results have both a textual syntax and an XML syntax.[5]

2.1 Mathematical Background for MathQL-1 Operational Semantics

We will present MathQL-1 semantics in a natural operational style [8] and we will use a simple type system that includes basic types for strings and booleans, plus some type constructors such as product and exponentiation. $y : Y$ will represent a typing judgement.

Primitive types. The type of boolean values will be denoted by `Boole` and its elements are `T` and `F`.
The type of strings will be denoted by `String` and its elements are the finite sequences of characters. Grammatical productions, represented as strings in angle brackets, will be used to denote the subtype of `String` containing the produced sequences of characters.

Type constructors. $Y * Z$ denotes the product of the types Y and Z whose elements are the ordered pairs (y, z) such that $y : Y$ and $z : Z$. The notation is also extended to a ternary product.

$Y \to Z$ denotes the type of functions from Y to Z and $f\ y$ denotes the application of $f : Y \to Z$ to $y : Y$. Relations over types, such as equality, are seen as functions to `Boole`.

`Listof` Y denotes the type of lists (i.e. ordered finite sequences) over Y. We will use the notation $[y_1, \cdots, y_m]$ for the list whose elements are $y_1, \cdots, y_m$.

`Setof` Y denotes the type of finite sets (i.e. unordered finite sequences without repetitions) over Y. With this constructor we can give a formal meaning to most of the standard set-theoretical notation. For instance we will use the following:

[5] In a distributed setting where query engines are implemented as stand-alone components, both queries and query results must travel inside the system so both need to be encoded in clearly defined format with a rigorous semantics.

- $\subseteq$: $(\texttt{Setof}\ Y) \to (\texttt{Setof}\ Y) \to \texttt{Boole}$ (infix)
- $\between$: $(\texttt{Setof}\ Y) \to (\texttt{Setof}\ Y) \to \texttt{Boole}$ (infix)
- $\sqcup$: $(\texttt{Setof}\ Y) \to (\texttt{Setof}\ Y) \to (\texttt{Setof}\ Y)$ (the disjoint union, infix)

$U \between W$ means $(\exists u \in U)\ u \in W$ and expresses the fact that $U \cap W$ is inhabited as a primitive notion, i.e. without mentioning intersection and equality as for $U \cap W \neq \emptyset$, which is equivalent but may be implemented less efficiently in real cases[6]. $U \between W$ is a natural companion of $U \subseteq W$ being its logical dual (recall that $U \subseteq W$ means $(\forall u \in U)\ u \in W$) and is already being used successfully in the context of a constructive (i.e. intuitionistic and predicative) approach to point-free topology[7].

Sets of couples play a central role in our model and in particular we will use:

- $\texttt{Fst} : (Y \times Z) \to Y$ such that $\texttt{Fst}\ (y, z) = y$.
- $\texttt{Snd} : (Y \times Z) \to Z$ such that $\texttt{Snd}\ (y, z) = z$.
- with the same notation, if W contains just one couple whose first component is y, then $W(y)$ is the second component of that couple. In the other cases $W(y)$ is not defined. This operator has type $(\texttt{Setof}\ (Y \times Z)) \to Y \to Z$.
- Moreover $W[y \leftarrow z]$ is the set obtained from W removing every couple whose first component is y and adding the couple (y, z). The type of this operator is: $(\texttt{Setof}\ (Y \times Z)) \to Y \to Z \to (\texttt{Setof}\ (Y \times Z))$.
- Also $U + W$ is the union of two sets of couples in the following sense:

$$\begin{gathered} U + \emptyset \text{ rewrites to } U \\ U + (W \sqcup \{(y, z)\}) \text{ rewrites to } U[y \leftarrow z] + W \end{gathered}$$

The last three operators are used to read, write and join association sets, which are sets of couples such that the first components of two different elements are always different. These sets will be exploited to formalize the memories appearing in evaluation contexts.

Now we are able to type the main objects needed in the formalization:

- An attribute name a is of type $\texttt{String}$.
- A multiple string value V is an object of type $T_0 = \texttt{Setof String}$.
- A set of attributes A is an association set connecting the attribute names to their values, therefore its type is $T_1 = \texttt{Setof}\ (\texttt{String} \times T_0)$.
- A reference r is an object of type $\texttt{String}$.
- The type of a collection D of the sets of attributes of a resource is $T_2 = \texttt{Setof}\ T_1$.
- A resource is a reference with its sets of attributes, so its type is $T_3 = \texttt{String} \times T_2$.
- A set of resources S is an association set of type $T_4 = \texttt{Setof}\ T_3$.
- A path name s is a non-null list of strings with the first component highlighted therefore its type is $T_5 = \texttt{String} \times \texttt{Listof String}$.

[6] As for $\phi \vee \psi$ which may have a more efficient implementation than $\neg(\neg\phi \wedge \neg\psi)$.

[7] Sambin G., Gebellato S. *A preview of the basic picture: a new perspective on formal topology.* Lecture Notes in Computer Science n.1657 p.194-207. Springer, 1999.

– The sets B in the triples of relations are association sets of type `Setof`$(T_5 \times T_0)$.

We will also need some primitive functions that mostly retrieve the informations that an implemented query engine obtains reading its underlying database. These functions are `Relation`, `Pattern`, `Property`, `Unquote` and will be explained when needed.

2.2 Textual Syntax and Semantics of MathQL-1 Queries

MathQL-1 query expressions, or simply expressions, fall into three categories.

– Expressions denoting a set of resources: belong to the grammatical production `<mq_set>`. Their semantics is given by the infix evaluating relation $\Downarrow_s$.
– Expressions denoting a boolean condition: are evaluated by the infix relation $\Downarrow_b$ and belong to the grammatical production `<mq_boole>`.
– Expressions denoting a multiple string value: are evaluated by the infix relation $\Downarrow_v$ and belong to the grammatical production `<mq_val>`.

Expressions can contain quoted strings with the following syntax:

```
<string>      ::= '"' [ "\" . | '^ "\' ]* '"'
<path>        ::= <string> [ "/" <string> ]*
<string_list> ::= <string> [ "," <string> ]*
```

When these strings are unquoted, the surrounding double quotes are deleted and each character is not escaped (the escape character is the backslash). This operation is formally performed by the function `Unquote` of type `String` $\rightarrow$ `String`. Moreover `Name` : `<path>` $\rightarrow T_5$ is an helper function that converts a linearized path name in its structured format. Formally `Name` $(q_0 \ / \ q_1 \ / \ \cdots \ / \ q_m)$ rewrites to (`Unquote` i_0, [`Unquote` $i_1, \cdots,$ `Unquote` i_m]).

Expressions can also contain variables for single resources (rvar), variables for sets of resources, i.e. for query results (svar) and variables for multiple strings values (vvar).

```
<alpha>  ::= [ 'A - Z' | 'a - z' | ':_' ]+
<number> ::= [ '0 - 9' ]+
<id>     ::= <alpha> [ <alpha> | <number> ]*
<rvar>   ::= "@" <id>
<svar>   ::= "%" <id>
<vvar>   ::= "$" <id>
```

Expressions are evaluated in a context $\Gamma = (\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$ which is a quadruple of association sets which connect svar's to sets of resources, rvar's to resources, rvar's to sets of attributes and vvar's to multiple string values[8]. Therefore the type K of the context Γ is:

[8] Γ_a is an auxiliary context used for `ex` and `''dot''` constructions (see below).

$$\texttt{Setof (<svar>} \times T_4) \times \texttt{Setof (<rvar>} \times T_3) \times$$
$$\times\ \texttt{Setof (<rvar>} \times T_1) \times \texttt{Setof (<vvar>} \times T_0)$$

and the three evaluating relations are of the following types:

$$\Downarrow_s : (K \times \texttt{<mq\_set>}) \to T_4 \to \texttt{Boole},$$
$$\Downarrow_b : (K \times \texttt{<mq\_boole>}) \to \texttt{Boole} \to \texttt{Boole},$$
$$\Downarrow_v : (K \times \texttt{<mq\_val>}) \to T_0 \to \texttt{Boole}.$$

Expressions denoting a set of resources. These expressions denote queries or subqueries.

```
<refine>    ::= [ "sub" | "super" ]?
<qualifier> ::= [ "inverse" ]? <refine> <path>
<assign>    ::= <vvar> "<-" <path>
<attr_list> ::= [ "attr" <assign> [ "," <assign> ]* ]?
<mq_set>    ::= "ref" <mq_val> | "pattern" <mq_val>
              | <svar> | <rvar> | "(" <mq_set> ")"
              | "relation" <qualifier> <mq_val> <attr_list>
              | "select" <rvar> "in" <mq_set> "where" <mq_boole>
              | <mq_set> [ "union" | "intersect" | "diff" ] <mq_set>
              | "let" <svar> "be" <mq_set> "in" <mq_set>
              | "let" <vvar> "be" <mq_val> "in" <mq_set>
```

intersect, *union*, *diff* are left-associative and have decreasing precedence order.

– The `let` constructions introduce svar's and vvar's in the context; formally:

$$\frac{i : \texttt{<svar>} \quad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), x_1) \Downarrow_s S_1 \quad ((\Gamma_s[i \leftarrow S_1], \Gamma_r, \Gamma_a, \Gamma_v), x_2) \Downarrow_s S_2}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), \text{let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_s S_2}$$

$$\frac{i : \texttt{<vvar>} \quad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), x_1) \Downarrow_v V \quad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v[i \leftarrow V]), x_2) \Downarrow_s S}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), \text{let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_s S}$$

– Then we have some constructions for reading variables and grouping:

$$\frac{i : \texttt{<svar>}}{(\Gamma, i) \Downarrow_s \Gamma_s(i)} \qquad \frac{i : \texttt{<rvar>}}{(\Gamma, i) \Downarrow_s \{\Gamma_r(i)\}} \qquad \frac{(\Gamma, x) \Downarrow_s S}{(\Gamma, (x)) \Downarrow_s S}$$

$\Gamma_s(i)$ and $\{\Gamma_r(i)\}$ mean $\emptyset$ if i is not defined and Γ is $(\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$.

– The `union` construction "sums" two sets of resources in the following way:

$$\frac{(\Gamma, x_1) \Downarrow_s S_1 \quad (\Gamma, x_2) \Downarrow_s S_2}{(\Gamma, x_1 \text{ union } x_2) \Downarrow_s S_1 \oplus S_2}$$

where $\oplus$ puts together the sets of attributes belonging to the same resource:

1 $(S_1 \sqcup \{(r, D_1)\}) \oplus (S_2 \sqcup \{(r, D_2)\})$ rewrites to $S_1 \oplus S_2 \oplus \{(r, D_1 \cup D_2)\}$
2 $S_1 \oplus S_2$ rewrites to $S_1 \cup S_2$

$\oplus$ is defined associative and rule 1 takes precedence over rule 2.

– The `intersect` construction "multiplies" two sets of resources in this way:

$$\frac{(\mathit{\Gamma}, x_1) \Downarrow_s S_1 \quad (\mathit{\Gamma}, x_2) \Downarrow_s S_2}{(\mathit{\Gamma}, x_1 \text{ intersect } x_2) \Downarrow_s S_1 \otimes S_2}$$

where $\otimes$ builds the product of the sets of attributes belonging to the same resource. Here $D_1 \times D_2 = \{A_1 \oplus A_2 \mid A_1 \in D_1, A_2 \in D_2\}$.

$$\begin{array}{lr} 1 & (S_1 \sqcup \{(r, D_1)\}) \otimes (S_2 \sqcup \{(r, D_2)\}) \text{ rewrites to } (S_1 \otimes S_2) \cup \{(r, D_1 \times D_2)\} \\ 2 & S_1 \otimes S_2 \text{ rewrites to } \emptyset \end{array}$$

Rule 1 takes precedence over rule 2.

– The `diff` construction makes the "difference" of two sets of resources:

$$\frac{(\mathit{\Gamma}, x_1) \Downarrow_s S_1 \quad (\mathit{\Gamma}, x_2) \Downarrow_s S_2}{(\mathit{\Gamma}, x_1 \text{ diff } x_2) \Downarrow_s S_1 \ominus S_2}$$

where $A \ominus B$ gives the resources of A, with their attributes, that don't belong to B (without considering the associated attributes):

$$\begin{array}{lr} 1 & (S_1 \sqcup \{(r, D_1)\}) \ominus (S_2 \sqcup \{(r, D_2)\}) \text{ rewrites to } S_1 \ominus S_2 \\ 2 & S_1 \ominus S_2 \text{ rewrites to } S_1 \end{array}$$

Again rule 1 takes precedence over rule 2.

– The `ref` (reference) construction turns a given set of references into a set of resources, each without attributes (i.e makes a coercion between the types T_0 and T_4); formally:

$$\frac{(\mathit{\Gamma}, x) \Downarrow_v V}{(\mathit{\Gamma}, \text{ref } x) \Downarrow_s \{(v, \emptyset) \mid v \in V\}}$$

– The `pattern` construction gives the set of resources whose reference matches at least one of the given POSIX 1003.2-1992 (included in POSIX 1003.1-2001[9]) regular expressions, each without attributes:

$$\frac{(\mathit{\Gamma}, x) \Downarrow_v V}{(\mathit{\Gamma}, \text{pattern } x) \Downarrow_s \{(r, \emptyset) \mid (\exists v \in V) \ r \in \texttt{Pattern } v\}}$$

where `Pattern` v represents the set of references matching the regular expression v, among the ones available in the underlying database.

– The `relation` construction regards RDF properties for building the set of the resources (r) in the specified relation $\{(v, r, B)\}$, its sub-relations (`sub`) or its super-relations (`super`) with some resource in a given set or with multiple string values in general. Each of these have the set of attributes defined by the assignments following the `attr` clause. In this rule D_2 is $\{\texttt{Assign } B \ \{q_1, \cdots, q_n\}\}$ and R is `Relation` f (`Name` p):

$$\frac{f : \texttt{<refine>} \quad p : \texttt{<path>} \quad (\mathit{\Gamma}, x) \Downarrow_v V \quad q_1 : \texttt{<assign>} \quad \ldots \quad q_n : \texttt{<assign>}}{(\mathit{\Gamma}, \text{relation } f \ p \ x \text{ attr } q_1, \cdots, q_n) \Downarrow_s \bigoplus \{\{(r, D)\} \mid (\exists v \in V) \ (v, r, B) \in R\}}$$

[9] `<http://www.unix-systems.org/version3/ieee_std.html>`.

`Assign` builds a set of (eventually compound) attributes whose values are fetched from B using the correspondences given by $q_1, \cdots, q_n$. Formally if i : `<vvar>` and p : `<path>`, we define `Assign` $B\ Q = \bigoplus\{(i, B(\texttt{Name}\ p)) \mid (i \leftarrow p) \in Q\}$. `Relation` $f\ s$ gives the set of triples (R) of the relation s.
If $s = (s_0, [s_1, \cdots, s_m])$, `Relation` gives the set of triples of the relation obtained by the composition of the relations $s_0, s_1, \cdots, s_m$. This is the case of a structured property s_0 that has (i.e. is described by) another property s_1 and so on. If f is `sub` or `super`, `Relation` gives also the set of triples of the sub-relations or super-relations of s.
If the `inverse` switch is present, the inverse relation must be used in place of the direct one: while the argument x represents the subject of the property specified by `relation`, on the contrary x represents the object of the property specified by `relation inverse`.
Formally replace R with $R' = \{(r, v, B) \mid (v, r, B) \in R\}$ in the previous rule.
Finally note that the vvar's introduced by `relation` differ from the ones introduced by the `let` clause even if they have the same name.

– The `select` construction extracts from a given set of resources the ones meeting a given boolean condition. The condition is tested for each resource of the set and the resources are assigned to a given rvar before the test. Formally we have:

$$\frac{i : \texttt{<rvar>} \quad (\Gamma, x_1) \Downarrow_s S \quad x_2 : \texttt{<mq\_boole>}}{(\Gamma, \text{select } i \text{ in } x_1 \text{ where } x_2) \Downarrow_s \texttt{Select}\ S\ i\ \Gamma\ x_2}$$

where the `Select` function is defined below.

$$\frac{i : \texttt{<rvar>} \quad x : \texttt{<mq\_boole>}}{\texttt{Select}\ \emptyset\ i\ \Gamma\ x \text{ rewrites to } \emptyset}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_s, \Gamma_r[i \leftarrow R], \Gamma_a, \Gamma_v), x) \Downarrow_b \texttt{F}}{\texttt{Select}\ (S \sqcup \{R\})\ i\ \Gamma\ x \text{ rewrites to } \texttt{Select}\ S\ i\ \Gamma\ x}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_s, \Gamma_r[i \leftarrow R], \Gamma_a, \Gamma_v), x) \Downarrow_b \texttt{T}}{\texttt{Select}\ (S \sqcup \{R\})\ i\ \Gamma\ x \text{ rewrites to} (\texttt{Select}\ S\ i\ \Gamma\ x) \cup \{R\}}$$

where R is actually a couple of the form (r, D) and $\Gamma = (\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$.

Expressions denoting a boolean condition. These expressions appear in the `where` clause of a `select` construction.

```
<mq_boole> ::= "false" | "true" | "(" <mq_boole> ")"
             | "not" <mq_boole> | "ex" <mq_boole>
             | <mq_boole> [ "and" | "or" ] <mq_boole>
             | <mq_val> [ "sub" | "meet" | "eq" ] <mq_val>
```

and and *or* are left-associative. The precedence (high to low) is: *not*, *and*, *or*, *ex*.

– We have constructions for constants, standard operations and grouping:

$$\frac{}{(\Gamma, \text{false}) \Downarrow_b \mathtt{F}} \quad \frac{}{(\Gamma, \text{true}) \Downarrow_b \mathtt{T}} \quad \frac{(\Gamma, x) \Downarrow_b b}{(\Gamma, (x)) \Downarrow_b b} \quad \frac{(\Gamma, x) \Downarrow_b \mathtt{T}}{(g, \text{not } x) \Downarrow_b \mathtt{F}} \quad \frac{(\Gamma, x) \Downarrow_b \mathtt{F}}{(g, \text{not } x) \Downarrow_b \mathtt{T}}$$

$$\frac{(\Gamma, x_1) \Downarrow_b \mathtt{F}}{(\Gamma, x_1 \text{ and } x_2) \Downarrow_b \mathtt{F}} \quad \frac{(\Gamma, x_1) \Downarrow_b \mathtt{T} \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1 \text{ and } x_2) \Downarrow_b b}$$

$$\frac{(\Gamma, x_1) \Downarrow_b \mathtt{T}}{(\Gamma, x_1 \text{ or } x_2) \Downarrow_b \mathtt{T}} \quad \frac{(\Gamma, x_1) \Downarrow_b \mathtt{F} \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1 \text{ or } x_2) \Downarrow_b b}$$

The binary operations are evaluated with an early-out (C-style) strategy.

– The `ex` (exists) construction gives access to the sets of attributes associated to the resources in the Γ_r part of the context and does this by loading its Γ_a part, which is used by the `<rvar>.<vvar>` construction (see below). `ex` is true if the condition following it is satisfied by at least one pool of attribute sets, one for each resource in the Γ_r part of the context. Formally we have the rule, where $\mathtt{All}\ \Gamma_r = \{\Delta_a \mid \Delta_a(i) = A \text{ iff } A \in \mathtt{Snd}\ \Gamma_r(i)\}$ and Δ_a has the type of Γ_a:

$$\frac{}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), \text{ex } x) \Downarrow_b ((\exists \Delta_a \in \mathtt{All}\ \Gamma_r)\ ((\Gamma_s, \Gamma_r, \Gamma_a + \Delta_a, \Gamma_v), x) \Downarrow_b \mathtt{T})}$$

– The `sub`, `meet` and `eq` constructions compare two sets of string values. The comparisons are extensional and the string equality is case-sensitive.

$$\frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1 \text{ sub } x_2) \Downarrow_b (V_1 \subseteq V_2)} \quad \frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1 \text{ meet } x_2) \Downarrow_b (V_1 \between V_2)}$$

$$\frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1 \text{ eq } x_2) \Downarrow_b (V_1 = V_2)}$$

The `eq` operator is introduced because the evaluation of x_1 eq x_2 may be more efficient than that of x_1 sub x_2 and x_2 sub x_1.
As an application of the `sub` and `meet` operators, consider a set of resources denoted by X : `<mq_set>` and a boolean condition denoted by B : `<mq_boole>` and depending on the variable @u : `<rvar>`. The test "is B satisfied for each resource of X?" is expressed by "refof X sub refof select @u in X where B" whereas the dual test "is B satisfied for some resource of X?" is expressed by "refof X meet refof select @u in X where B".
`refof` makes a coercion between the types T_4 and T_0 (see below).

Expressions denoting a multiple string value. These expressions appear as operands of the `sub`, `meet` and `eq` construction.

```
<mq_val> ::= "{" [ <string_list> ]? "}" | <string>
         | <rvar> "." <vvar> | <vvar> | "(" <mq_val> ")"
         | "refof" <mq_set> | "property" <qualifier> <mq_val>
```

– A set of strings can be given explicitly with these two constructions:

$$\frac{q_1 : \texttt{<string>} \quad \cdots \quad q_m : \texttt{<string>}}{(\Gamma, \{q_1, \cdots, q_m\}) \Downarrow_v \{\texttt{Unquote}\ q_1, \cdots, \texttt{Unquote}\ q_m\}} \qquad \frac{q : \texttt{<string>}}{(\Gamma, q) \Downarrow_v \{\texttt{Unquote}\ q\}}$$

– `refof` (references of) allows to obtain a set of strings gathering the references in a set of resources (i.e. it makes a coercion between the types T_4 and T_0):

$$\frac{(\Gamma, x) \Downarrow_s S}{(\Gamma, \text{refof}\ x) \Downarrow_v \{\texttt{Fst}\ u \mid u \in S\}}$$

– We have a construction for grouping and two for reading variables: one reads the Γ_a part of the context which is updated by the `ex` clause, while the other read the Γ_v part of the context is updated by the `let` clause for vvar's:

$$\frac{(\Gamma, x) \Downarrow_v V}{(\Gamma, (x)) \Downarrow_v V} \qquad \frac{i : \texttt{<rvar>} \quad j : \texttt{<vvar>}}{(\Gamma, i.p) \Downarrow_v \Gamma_a(i)(j)} \qquad \frac{i : \texttt{<vvar>}}{(\Gamma, i) \Downarrow_v \Gamma_v(i)}$$

where $\Gamma = (\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$. $\Gamma_a(i)(j)$ and $\Gamma_v(i)$ mean $\emptyset$ if i or j are not defined. With the "dot" construction a vvar introduced by a `relation` (i.e. an attribute) can be read only specifying an associated rvar (i.e. a resource) but this restriction offers some advantages: it conforms to the general idea of treating an attribute as an entity which is always related to a resource and allows an unambiguous read of those attributes related to different resources but sharing the same name. Note that the use of the `let` construction may produce unavoidable attribute name collisions in the scope of nested `where` clauses as in the following sample query where `...` is a place holder:

```
let %s be relation "..." "..." attr $a <- "a" in
select @u1 in %s where "..." sub refof
   select @u2 in %s where ex @u1.$a sub @u2.$a
```

– Finally a set of strings can be obtained reading the values of a named property, its sub-properties (`sub`) or its super-properties (`super`) of a given set of string arguments (i.e. references subjects of the specified property) with a semantics which is very close to that of the `relation` construction: `property` regards RDF properties to build multiple string values for checking and filtering metadata information.

$$\frac{f : \texttt{<refine>} \quad p : \texttt{<path>} \quad (\Gamma, x) \Downarrow_v V}{(\Gamma, \text{property}\ f\ p\ V) \Downarrow_v \{v \mid (\exists r \in V)\ (r, v) \in Q\}}$$

where $Q = \texttt{Property}\ f\ (\texttt{Name}\ p)$ and $\texttt{Property}\ f\ s$ gives the set of couples of the property whose name is s, representing the values of the instances of the property attributed to the item referenced in the first element of a couple. If $s = (s_0, [s_1, \cdots, s_m])$, `Property` gives the set of couples of the property obtained by the composition of the properties $s_0, \cdots, s_n$. This is the case of structured values of a property s_0 that has (i.e. is described by) another

property s_1 and so on. If f is `sub` or `super`, `Property` gives also the couples respectively of the sub or super-properties of s.
If the `inverse` switch is present, the inverse function must be used in place of the direct one: while the argument x represents the subject of the property specified by `property`, on the contrary x represents the object of the property specified by `property inverse`.
Formally replace Q with $Q' = \{(v, r) \mid (r, v) \in Q\}$ in the previous rule.

2.3 Textual Syntax and Semantics of MathQL-1 Query Results

The textual representations of query results belong to the grammatical production `<mqr_set>` whose semantics is described by four (infix) evaluating relations: $\Rightarrow_a$, $\Rightarrow_g$, $\Rightarrow_r$ and $\Rightarrow_s$.
$\Rightarrow_a$: `<mqr_attr>` $\rightarrow$ (`String` $\times T_0$) $\rightarrow$ `Boole` evaluates an attribute with a multiple string value. $\Rightarrow_g$: `<mqr_group>` $\rightarrow T_1 \rightarrow$ `Boole` evaluates a set of attributes with a their values. $\Rightarrow_r$: `<mqr_res>` $\rightarrow T_3 \rightarrow$ `Boole` evaluates resource with its sets of attributes. $\Rightarrow_s$: `<mqr_set>` $\rightarrow T_4 \rightarrow$ `Boole` evaluates a set of resources with their sets of attributes.
Note that a multiple string value can be empty. In fact, in an RDF data model a property can be optionally used, even if it is always declared in an RDF schema. Also note that the sets of attributes are always inhabited.

```
<mqr_attr>  ::= <vvar> [ "=" <string_list> ]?
<mqr_group> ::= "{" <mqr_attr> [ ";" <mqr_attr> ]* "}"
<groups>    ::= <mqr_group> [ "," <mqr_group> ]*
<mqr_res>   ::= <string> [ "attr" <group> ]?
<mqr_set>   ::= [ <mqr_res> [ ";" <mqr_res> ]* ]?
```

Formally the evaluation works as follows:

$$\frac{i : \texttt{<vvar>}\ q_1 : \texttt{<string>}\ \ldots\ q_m : \texttt{<string>}}{i = q_1, \ldots, q_m \Rightarrow_a (i, \{\texttt{Unquote}\ q_1, \ldots, \texttt{Unquote}\ q_m\})} \qquad \frac{x_1 \Rightarrow_a a_1\ \ldots\ x_m \Rightarrow_a a_m}{\{x_1; \ldots; x_m\} \Rightarrow_g \{a_1, \ldots, a_m\}}$$

$$\frac{q : \texttt{<string>} \quad x_1 \Rightarrow_g A_1 \quad \cdots \quad x_m \Rightarrow_g A_m}{q \text{ attr } x_1, \cdots, x_m \Rightarrow_r (\texttt{Unquote}\ q, \{A_1, \cdots, A_m\})} \qquad \frac{x_1 \Rightarrow_r r_1 \quad \cdots \quad x_m \Rightarrow_r r_m}{x_1; \cdots; x_m \Rightarrow_s \{r_1, \cdots, r_m\}}$$

3 Implementation and Testing

3.1 Implementation

In this section we will briefly discuss the implementation of a MathQL-1 querying engine in the context of the HELM project describing the functions and relations available from the HELM metadata schemas[10] as well as other issues concerning the underlying database management system and the implemented software. Currently HELM metadata provide the following information on the mathematical resources of the library.

[10] `<http://www.cs.unibo.it/helm/schemas/schema-h>`,
`<http://www.cs.unibo.it/helm/schemas/schema-hth>`.

- The standard Dublin Core metadata properties[11].
- The list of objects the object depends on. This information is available through a relation named `refObj`.
- The list of objects depending on the object. This information is available through a relation named `backPointer`.
- An alias for the reference of the object, given by the function `shortName`.

HELM is testing PostgreSQL[12] DBMS and Galax XQuery engine[13] as the support for the querying engine. The querying engine is written in Caml[14] for an easy integration with the other software developed for the HELM project, and currently it consists of the following parts.

- The *interpreter*, implemented by D. Lordi [2] and now re-implemented by L. Natile, executes a query given in its Caml representation (as a suitable data structure) giving back a Caml representation of the query result.
- The *input/output utilities* interface the textual or XML representation of a query or of a query result with its internal Caml representation.
- The *query generator* is the interface between the interpreter and the HELM proof assistant which needs to search the library for various purposes like interactive or automated proof searching. In particular the proof assistant provides a command (or tactic) named *SearchPatternApply* which uses the generator to issue a query for the set of statements that can possibly refine the current goal.

What follows is the textual representation of the query issued by the generator when the HELM proof assistant *SearchPatternApply* command is applied to the goal $2 * m \leq 2 * n$ where m and n are natural numbers and 2 is actually the successor of the successor of 0.
The references appearing in this query denote the following HELM resources:

"cic:/Coq/Init/Peano/le.ind#1/1"	less or equal
"cic:/Coq/Init/Peano/mult.con"	multiplication
"cic:/Coq/Init/Datatypes/nat.ind#1/1/2"	successor
"cic:/Coq/Init/Datatypes/nat.ind#1/1/1"	zero

```
let $positions be {"MainConclusion", "InConclusion"} in
let $universe be {"cic:/Coq/Init/Datatypes/nat.ind#1/1/1",
                  "cic:/Coq/Init/Peano/mult.con",
                  "cic:/Coq/Init/Datatypes/nat.ind#1/1/2",
                  "cic:/Coq/Init/Peano/le.ind#1/1"} in
 select @uri0 in
   select @uri in relation inverse "refObj"
    "cic:/Coq/Init/Peano/le.ind#1/1"
```

[11] `<http://purl.org/dc/elements/1.1/>`.
[12] `<http://www.postgresql.org>`.
[13] `<http://db.bell-labs.com/galax/>`.
[14] `<http://caml.inria.fr>`.

```
    attr $pos <- "position"
   where ex "MainConclusion" sub @uri.$pos
 intersect
   select @uri in relation inverse "refObj"
    "cic:/Coq/Init/Peano/mult.con"
    attr $pos <- "position" where ex "InConclusion" sub @uri.$pos
 intersect
   select @uri in relation inverse "refObj"
    "cic:/Coq/Init/Datatypes/nat.ind#1/1/2"
    attr $pos <- "position" where ex "InConclusion" sub @uri.$pos
 intersect
   select @uri in relation inverse "refObj"
    "cic:/Coq/Init/Datatypes/nat.ind#1/1/1"
    attr $pos <- "position" where ex "InConclusion" sub @uri.$pos
where
   refof select @uri in relation "refObj" refof @uri0
           attr $pos <- "position"
           where ex $positions meet @uri.$pos
   sub $universe
```

The main operation in the query is the `select @uri0` and its `in` clause builds, through three `intersect`, the set of statements having each of these resources in their conclusions (in an appropriate position). The `where` clause is responsible for filtering out the statements whose conclusion refers to other resources than the ones above. Also note the exploitation of the `inverse` switch to invert the `refObj` relation. This query has some optimizations: the `let` constructions evaluate the sets `$positions` and `$universe` (this contains the resources that must appear in the statement), outside the `where` clause, which is evaluated many times during the execution of the `select`. Moreover the test:

```
$positions meet @uri.$pos
```

should be more efficient than the equivalent test:

```
"MainConclusion" sub @uri.$pos or "InConclusion" sub @uri.$pos
```

as this involves a larger number of operators.

3.2 Testing

The information currently available in the PostgreSQL database concerns 15244 HELM objects and comes from the scan of 416699 RDF statements stored in 80 Mb of disk space[15] while the database itself uses 382 Mb of disk space.

The following test evaluates the total processing time (including the query building time) of 165 queries (issued by the generator and processed by the Natile interpreter) like the one in the query example (see 3.1), with respect to the query complexity, which is proportional to the size of the `%universe` set.

[15] See [2] for a description of how the database is built from the RDF files.

The first table concerns PostgreSQL, the second table concerns Galax. This test shows that Natile engine handles complex queries more efficiently.

size	**issued queries**	**time/size (mean)**	**time/size (variance)**
1 to 2	59	0.25 sec.	0.23 sec.
3 to 9	106	0.03 sec.	0.02 sec.
1 to 9	165	0.11 sec.	0.17 sec.

size	**issued queries**	**time/size (mean)**	**time/size (variance)**
1 to 2	59	13.00 sec.	13.53 sec.
3 to 9	106	0.33 sec.	0.23 sec.
1 to 9	165	4.86 sec.	10.12 sec.

References

1. Asperti A., Padovani L., Sacerdoti Coen C., Guidi F., Schena I.: Mathematical Knowledge Management in HELM. In Electronic Proc. of MKM 2001. `<http://www.emis.de/proceedings/MKM2001/>`
2. Lordi D.: Sperimentazione e Sviluppo di Strumenti per la gestione di metadati. M. Thesis in Computer Science, University of Bologna. Advisor: A. Asperti. 2002
3. Rayavarapu S.: W3C Query languages, 29 January 2001. `<http://www1.coe.neu.edu/~srayavar/W3CQL/ql.html>`
4. Lassila O. and others: Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February 1999. `<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>`
5. Brickley D. and others: RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft 30 April 2002. `<http://www.w3.org/TR/rdf-schema/>`
6. Schena I. *Towards a Semantic Web for Formal Mathematics.* Ph.D. Thesis in Computer Science, University of Bologna. Supervisor: A. Asperti, February 2002.
7. Berners-Lee T. and others: Uniform Resource Identifiers (URI): Generic Syntax (RFC 2396). `<http://www.ietf.org/rfc/rfc2396.txt>`
8. Winskel G.: The formal semantics of programming languages: an introduction. MIT Press Series in the Foundations of Computing. London: MIT Press, 1993
9. Bray T. and others: Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000. `<http://www.w3.org/TR/REC-xml/>`
10. Chamberlin D. and others: XQuery 1.0: An XML Query Language, W3C Working Draft 16 August 2002. `<http://www.w3.org/TR/xquery/>`