# tolog – a topic maps query language

Lars Marius Garshol

Ontopia AS, Oslo, Norway,
larsga@ontopia.net, http://www.ontopia.net

**Abstract.** This paper describes a query algebra for tolog, a query language for Topic Maps inspired by Prolog and very similar to Datalog. The language is based on binding variables by matching predicates against the topic map being queried, and contains predicates for querying any aspect of the Topic Maps Data Model (TMDM) [ISO13250-2], as well as support for user-defined predicates. SQL-like features like aggregate functions, projection, ordering, and result set paging are also supported. The paper uses a formal model for Topic Maps called Q to formally define the semantics of tolog. The standard tolog predicates are defined, together with a query algebra. This gives the query language a firm basis, enables interoperable implementations, and serves as the starting point for further work on the language.

## 1 Introduction

tolog is a query language for Topic Maps originally inspired by Prolog. It is much more similar to Datalog, however, in that no particular evaluation algorithm is required, there is no backtracking, the order of clauses is irrelevant, and complex terms cannot be arguments to predicates. The language supports standard Horn clauses, but also supports NOT, OR, and SQL features like projection, counting, sorting, and paging of result sets.

Predicates are of three kinds: user-defined (through predicate declarations), built-in (this includes comparison predicates as well as predicates based on TMDM), and dynamic predicates (created from occurrence and association types in the topic map).

The language has seen three independent implementations, and has been the foundation for several commercial Topic Maps applications. The language forms the core of the OKS, a commercial suite of Topic Maps tools from Ontopia, and is also implemented in TM4J, an open source Topic Maps engine. The third implementation is part of Concept Glossary Manager from Rodans [Strychowski05]. An effort has also begun to create a fourth Java implementation based on TMAPI by porting the TM4J implementation. It was also selected as the basis for the standard TMQL language to be defined by ISO.

### 1.1 Brief tutorial

In order to make this paper more self-contained we give a brief tolog tutorial here. For a fuller introduction to the language, see [Garshol05b].

Among the simplest possible tolog queries is:

```
instance-of($C, composer)?
```

This uses the built-in predicate `instance-of`, which relates types to their instances. In this case the second argument is the topic reference `composer` and the first is the variable `$C`. The query result is all values for `$C` that make the query true, that is, all instances of the topic type composer, or, informally, all composers.

Predicates can be chained with the AND operator, syntactically represented by comma, just as in Prolog and Datalog, so the following query would also give the birthdate for each composer:

```
instance-of($C, composer), birthdate($C, $D)?
```

As the comma translates to AND it follows that any composers which have no birthdate occurrence will not be included in the query results. Likewise, people other than composers which have a birthdate are not included.

Note that the `birthdate` predicate used above is actually a *dynamic* predicate, in the sense that it's an occurrence type in the topic map that becomes a predicate in tolog. The same happens with association types, as can be seen in the query below:

```
instance-of($C, composer), born-in($C : person, $P : place)?
```

This uses the `born-in` association type to give us the birthplace of the composer. `person` and `place` are association role types.

Another useful operator is the OR operator, which can be used as follows:

```
instance-of($C, composer), born-in($C : person, $P : place),
{ located-in($P : containee, norway : container) |
  located-in($P : containee, sweden : container) }?
```

This query finds all composers born in Norway or Sweden, and the place they were born. In some cases not all variables bound by the query are wanted, and in these cases the SELECT clause can be used to project down to only the wanted variables, as in this example:

```
select $C from
  instance-of($C, composer), born-in($C : person, $P : place),
  { located-in($P : containee, norway : container) |
    located-in($P : containee, sweden : container) }?
```

In this example only the composers will be returned by the query.

Another useful operator is the NOT operator, which makes it possible to find all matches which do not satisfy a particular condition, as shown in this example, which finds all composers not born in Italy:

```
select $C from
  instance-of($C, composer), born-in($C : person, $P : place),
  not(located-in($P : containee, italy : container))?
```

In the case where birth dates are not given for all composers we may still want to display it for those which have birth date, without losing the composers who do not have any. This can be done with the OPTIONAL construct, as follows:

```
  instance-of($C, composer), { birthdate($C, $D) }?
```

It's also possible to define new predicates, which can then be used in queries and also in the definition of still more predicates. This is how recursion is implemented in the language, and also how more complex queries can be written.

An example might be a predicate stating whether or not a person is Italian, which could be defined as follows:

```
  italian($C) :-
    instance-of($C, person),
    born-in($C : person, $P : place),
    located-in($P : containee, italy : container).
```

This predicate can now be used to find Italian composers, all Italians, every person who is not an Italian, etc etc.

In addition, tolog supports ordering the query result, as in the query below.

```
  instance-of($P, person) order by $P?
```

This would list all persons in alphabetical order. Each value type has its own ordering rules, which are used in the sorting. The `asc` and `desc` keywords can be used as in SQL. The same applies to `limit` and `offset`. So the following query:

```
  instance-of($P, person) order by $P limit 5?
```

would produce only the 5 first persons (ordered alphabetically).

So far, topics have only been referenced using IDs, which map to item identifiers in TMDM. However, it is considered best practice to refer to topics using subject identifiers, which are much more stable and reliable. Using this approach we could rewrite the first example as:

```
  instance-of($C, i"http://psi.ontopia.net/music/#composer")?
```

However, using topic references in this way can be difficult to read, especially when referencing occurrence and association types as predicate names. To simplify this prefix declarations can be used:

```
  using music for i"http://psi.ontopia.net/music/#"
  instance-of($C, music:composer)?
```

### 1.2 Related work

There is quite a variety of work that is related to tolog, falling in three main categories outlined below.

Work on Prolog has been going on since the early 1970s, and is still progressing. Datalog has likewise seen extensive work since 1978, especially in the late 80s and early 90s [Liu99]. tolog is only loosely connected with these languages, in that Prolog served as the initial inspiration, and the design was later found to be very similar to that of Datalog. The query algebra given here is not related to the formal semantics of Datalog in any way.

Several other query languages have been developed for Topic Maps [N0492], such as AsTMa?, Toma, TMPath, and TMRQL [Ahmed05]. These languages are quite varied, ranging from path-based languages, through SQL-inspired languages, functional languages, and even a SQL function library. An attempt was also made to show that Topic Maps queries could be implemented with XQuery [Robie01].

Query languages have also been developed for the W3C's RDF data model, and the present version of the query language that is currently being standardized by the W3C, called SPARQL [Seaborne05], is in many ways quite similar to tolog. It does variable matching in the same way, and supports projection, AND, OPTIONAL, and OR, but does not have NOT, predicate definitions, or ordering.

The key contribution of this paper is the formal definition of tolog, rather than the query language itself. Of the query languages discussed here only SPARQL can claim to have the same, although Robie's work as well as TMRQL were of course developed using formally defined languages.

### 1.3 The Q model

A formal definition of the semantics of a Topic Maps query language is impossible without a formal model of Topic Maps on which the query language can operate. This paper uses the Q model [Garshol05] as its foundation, since this is the only formal model for which there exists a defined mapping from TMDM. Given that tolog queries TMDM this was an absolute requirement.

Q represents Topic Maps as a set of five-tuples. The tuples can be thought of informally as a kind of extended RDF, with the following structure:

$(subject, property, identity, context, object)$

Here, *subject*, *property*, and *object* are as in RDF, *identity* is the identity of the statement, and *context* is the context in which the statement is considered true. In fact, the context is the identifier representing the set of topics making up the scope in a topic map. One difference with RDF, however, is that the values in the first four elements of a tuple can only be identifiers, which are propertyless objects used only as identifiers. Values, such as strings and URLs, are restricted to the last field.

More formally, a Q instance is a subset of $\mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{A}$ where $\mathcal{I}$ is the set of all identifiers (like blank nodes in RDF), $\mathcal{L}$ is the set of all values (strings etc), and $\mathcal{A} = \mathcal{L} \cup \mathcal{I}$.

In [Garshol05] a procedure for converting any TMDM instance into a Q instance is given, together with the reverse procedure, and also the same transformations for RDF models.

## 2 Query algebra

This section defines a query algebra that will be used in the next section to define the tolog language semantics. To do this, it is necessary to introduce some new concepts.

A variable is a token used in a query to identify a particular unknown value in a match to the query. Variables are written as upper-case identifiers preceded by a dollar sign: `$A`. The set of all variables is $\mathcal{V}$.

A *match* to a query is a set of tuples, where the first element of each tuple is a variable and the second is the value the variable is bound to in that match. More formally, the set of all matches is known as $\mathcal{M}$, and defined as follows:

$\mathcal{M} = \{m \in \mathcal{V} \times \mathcal{A} | \nexists k, v_1, v_2 : (k, v_1) \in m \wedge (k, v_2) \in m \wedge v_1 \neq v_2\}$

The function $vars : \mathcal{M} \rightarrow \mathcal{V}$ is defined as:

$vars(m) = \{k | \exists v : (k, v) \in m\}$

The function $val : \mathcal{V} \rightarrow \mathcal{A}$ is defined as:

$$val(k) = \begin{cases} v & \exists v | (k, v) \in m \\ null & \text{otherwise} \end{cases} \quad (1)$$

In the query algebra query results are represented by *match sets*, which are sets of matches. The set of all match sets is $\mathcal{S} = 2^{\mathcal{M}}$.

### 2.1 Predicates

Predicates are represented in the query algebra by functions which take the Q instance representing the topic map as the first argument and an argument tuple as the second argument.

### 2.2 The $\oplus$ operator

The $\oplus$ operator combines match sets and is consistent with the semantics of the AND operation. To define it we first define the concept of two matches being *compatible*. Two matches are compatible if they do not contradict each other; that is, they do not contain different values for the same variable.

Formally there is a relation $\sim$ over $\mathcal{M}$, such that:

$m_1 \sim m_2 \Leftrightarrow \nexists k, v_1, v_2 | (k, v_1) \in m_1 \wedge (k, v_2) \in m_2 \wedge v_1 \neq v_2$

The negation, $m_1 \not\sim m_2$, means that the two matches are not compatible; that is, they contradict each other.

Using this concept we can define the $\oplus$ operator:

$M_1 \oplus M_2 = \{m_1 \cup m_2 | \exists m_1 \in M_1, m_2 \in M_2 \wedge m_1 \sim m_2\}$

### 2.3 The $\odot$ operator

The $\odot$ operator combines match sets in a way that matches the semantics of the OPTIONAL operation. The formal definition is:

$$M_1 \odot M_2 = \{m_1 | m_1 \in M_1 \wedge \nexists m_2 \in M_2 : m_1 \subset m_2\} \cup \{m_2 | m_2 \in M_2 \wedge \nexists m_1 \in M_1 : m_1 \subseteq m_2 \wedge \nexists m_1' \in M_1 : m_1' \not\sim m_2\}$$

### 2.4 Projection

The $\pi : \mathcal{M} \times 2^{\mathcal{V}} \rightarrow \mathcal{M}$ function does projection for an individual match and is defined as follows:

$\pi(m, s) = \{(k, v) \in m | k \in s\}$

The $\mathit{\Pi} : \mathcal{S} \times 2^{\mathcal{V}} \rightarrow \mathcal{S}$ function does projection for match sets and is defined as follows:

$\mathit{\Pi}(M, s) = \{m | \exists m' \in M : m = \pi(m', s)\}$

### 2.5 The $\kappa$ function

The $\kappa : \mathcal{M} \times \mathcal{V} \rightarrow \mathcal{M}$ function essentially does counting. However, to define it, some new concepts are necessary.

First, we need the concept of a partition of a match set by a variable, which is effectively a set of subsets (blocks) of the match set where each block has all matches in the match set whose only difference is their value for that variable. The function $P : \mathcal{M} \times \mathcal{V} \rightarrow \mathcal{S}$ produces the partition of a match set by a given variable, and is defined as follows:

$$P(M, k) = \{\ M' \subset M | \forall m_1, m_2 \in M' : \exists k \in vars(m_1) = vars(m_2) : \pi(m_1, vars(m_1) - k) = \pi(m_2, vars(m_2) - k)\}$$

For any block in a partition there is a match that represents the common subset which all matches in the block share. The function $c : \mathcal{S} \times \mathcal{V} \rightarrow \mathcal{M}$ produces the common subset for any block in a partition, and is defined as follows:

$c(M', k) = \{(k', v) | k \neq k' \wedge \forall m \in M' : (k', v) \in m\}$

Given these concepts we can define the counting function as follows:

$\kappa(M, k) = \{m | \exists M' \in P(M, k) : m = c(M', k) \cup \{(k, |M' - c(M', k)|)\}\}$

The cardinality computation to set the value of $k$ in the count may look strange; the rationale is to exclude the match where $k$ has no value, ie: the match that is the common subset. This ensures that OPTIONAL operator can be used to produce the variable being counted, and that when there is no value for $k$ the count becomes 0 instead of 1.

## 3 Language semantics

In this section we will define the semantics of tolog queries from the bottom up, starting with literals and eventually progressing to full queries. In each case, a mapping from the tolog query expressions to the query algebra will be given.

### 3.1 Variables, literals, and references

Variables are written in tolog as `$NAME`.

Two types of literals are supported: strings, written `"abc"`, and numbers, written in the usual fashion.

In addition, references to topic map objects are allowed. These can use several syntaxes, where the most common is simple ID reference, like `foo`. In each case the effect is the same: the reference evaluates to the topic map object referred to.

In the query algebra literals and topic map object references map to constants representing their values.

### 3.2 Predicate application

Predicate applications are uses of a predicate, where the predicate is supplied with an argument tuple. An example of this might be:

```
instance-of($A, person)
```

In the query algebra, each predicate is a function which given a Q instance and an argument tuple produces a match set binding the variables in the argument tuple. The query above would therefore translate to the following in the query algebra if applied to the topic map $Q$:

$instance - of(Q, (\$A, person))$

The result would be a set of matches where $\$A$ is bound to all person topics in $Q$.

In the syntax predicates can be referenced in the same ways as topics. The details of the different syntaxes and the scope rules are a little involved, and so we will only focus on the semantics here.

When parsing a predicate application, the predicate function for this application is produced as follows:

1. If there is a user-defined predicate with this name, that predicate is used.
2. If there is a built-in predicate with this name, that predicate is used.
3. Interpret the predicate reference as a topic reference, and find the referenced topic $t$.
4. If $t$ is an association type, produce a dynamic association predicate as defined in 3.8 on page 13.
5. If $t$ is an occurrence type, produce a dynamic occurrence predicate as defined in 3.8 on page 13.
6. If $t$ is a name type, produce a dynamic name predicate as defined in 3.8 on page 13.
7. If all else fails, use this predicate: $empty(Q, p) = \emptyset$

### 3.3 Predicate expressions

A predicate expression is formed by combining predicate expressions using the AND, OR, NOT, and OPTIONAL operators. The mappings of these to the query algebra is relatively straightforward.

**AND** The AND operator maps to the $\oplus$ query algebra operator, such that any predicate expression of the form given below

```
e1, e2, ..., en
```

maps to the following query algebra expression:
$e_1 \oplus e_2 \oplus ... \oplus e_n$

**OR** The mapping of OR is very straightforward: it maps to the $\cup$ operator. Given a predicate expression as follows:

```
{ e1 | e2 | ... | en }
```

the corresponding query algebra expression is:
$e_1 \cup e_2 \cup ... \cup e_n$

**NOT** The mapping of NOT is a little more involved than might be expected. To be able to map NOT we need two sets of variables: $V$ being the set of all variables used in the query outside the NOT, and $V'$ being the set of variables used in the NOT. Given this the predicate expression

```
not(e)
```

would translate into the query algebra as follows:
$\Pi(\beta(\mathcal{A}^{|V'|}, V) - e, V \cap V')$
This makes NOT produce all match sets for which $e$ is not true, then project these down to the variables used outside the NOT. In this translation NOT can produce infinite match sets, but only formulated in terms of variables also used elsewhere in the query.

**OPTIONAL** The mapping of OPTIONAL is relatively straightforward; given a predicate expression as follows:

```
e1, { e2 }
```

the corresponding query algebra expression is:
$e_1 \odot e_2$

### 3.4 Predicate definitions

Predicate definitions effectively use a parameter list and a predicate expression to define a predicate. In the query algebra each predicate definition becomes a function whose body is defined using the query algebra.

A predicate definition takes the form

```
name(parameters) :- predicate-expression .
```

In the query algebra, let the name be $n$, the parameter tuple $p$, and the predicate expression $e$. The difficulty here is that $e$ produces match sets using the variables internal to the defined predicate, which are usually different from those used in the predicate application. This means that we need to translate the match set from the internal variables to the external ones (and also filter the match set using the literals provided in the argument tuple). The function that does this is $\tau$, defined as follows, where $M$ is a match set and $a$ the argument tuple:

$$\tau(M,p,a) = \{\ m | \exists m' \in M : \forall i \in [1,2,3,...,|p|] : \\ ((p[i] \in \mathcal{V} \wedge (a[i], val(m', a[i])) \in m' \wedge (p[i], val(m, p[i])) \in m) \vee \\ (p[i] \in \mathcal{A} \wedge (a[i], p[i]) \in m')\}$$

Given this function we can define the function resulting from a predicate definition as:

$$n(Q,p) = \tau(\Pi(e,p), a)$$

### 3.5 Queries

The overall structure of a query is:

```
SELECT select-clause FROM
  predicate-expression
ORDER BY order-clause
LIMIT limit-clause
OFFSET offset-clause?
```

A query algebra expression $e_1$ for the `predicate-expression` is created as described above.

The next expression $e_2$ is produced as follows:

- If there is no `select-clause` then $e_2 = e_1$.
- If there is a `select-clause` containing the set of variables $s$ but no counted variables then $e_2 = \Pi(e_1, s)$.
- If there is a `select-clause` containing the set of variables $s$ and the counted variable is $k$ then $e_2 = \kappa(\Pi(e_1, s), k)$. (Only one counted variable is allowed.)

The `order-clause`, `limit-clause`, and `offset-clause` are not mapped to the query algebra, as these are relatively straightforward to understand, and have only a very limited impact on optimization.

### 3.6 Built-in comparison predicates

tolog has a number of comparison predicates which mirror those in other query and programming languages. These predicates are all reflections of infinite subsets of $\mathcal{A} \times \mathcal{A}$, and so are what the Datalog literature calls *unsafe*. This means that they cannot be used alone, as they do not sufficiently constrain the result set to guarantee that query results are not infinite in size.

The value sets are sets of tuples and so to define the predicate functions we need somehow to filter such sets based on the literals given in the predicate arguments, and then to produce a match set with variable bindings. This is done by the $\beta$ function. This function takes the predicate result set and a specification tuple, filters it with any literals given in the specification tuple, and produces a set of matches with each position in the n-tuples bound to any variables given in the specification tuple.

The $\beta$ function is defined as follows:

$\beta(R, s) = \{m | \exists t \in R : m = b(t, s) \wedge (\nexists i : s[i] \notin \mathcal{V} \wedge s[i] \neq t[i])\}$

The $b$ function is here a helper function which produces a match from a result tuple from a predicate, defined as:

$b(t, s) = \{(k, v) | \exists i : s[i] = k \neq * \wedge s[i] \in \mathcal{V} \wedge t[i] = v\}$

With this in hand we can define the predicate functions as follows:

$$
\begin{aligned}
= (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 = v_2\} \\
/ = (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 \neq v_2\} \\
< (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 < v_2\} \\
> (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 > v_2\} \\
<= (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 <= v_2\} \\
>= (Q, s) &= \beta(\{(v_1, v_2) | \exists v_1, v_2 \in \mathcal{A} \wedge v_1 >= v_2\}
\end{aligned}
$$

Note that in the syntax these predicates are infix predicates.

### 3.7 Built-in topic map predicates

tolog has a number of built-in predicates that are used to access the detailed structure of the topic map. They are only defined formally here, with no further explanation. More information on these predicates can be found in [Garshol05c].

**Supporting predicates** In order to define the built-in predicates some supporting predicates which are not visible in the language are needed. These are defined here.

The main supporting predicate is `_q`, which is formally defined as:

$q(Q, p) = \beta(Q, p)$

The `_self-or-supertype` predicate is easily defined:

```
_self-or-supertype($SUPER, $SUB) :- {
  xtm:superclass-subclas($SUPER :xtm:superclass, $SUB : xtm:subclass) |
  xtm:superclass-subclas($SUPER :xtm:superclass, $MID : xtm:subclass),
  _self-or-supertype($MID, $SUB) |
  $SUPER = $SUB
}.
```

The `_is_uri($LOC)` predicate is true for URIs. To define it we need the set of all URIs: $\mathcal{U} \subset \mathcal{A}$. Given that, the predicate function is easily defined:

$is - uri(Q, a) = \beta(\{(u)|u \in \mathcal{U}\}, p)$

The `_is-like` predicate is true for a pair of strings if they are similar. Precisely what this means is not defined, as this predicate is used for full-text search, and different full-text search systems have different definitions of similarity.

**Built-in predicates** Now that the supporting predicates are defined we can define the actual predicates. Note that the definition of these rules assume that in the mapping of TMDM to Q binary associations are not defined using templates (as in [Garshol05]), but instead in the same way as n-ary associations. This is necessary in order to provide association roles with their own identities.

```
using xtm for "http://www.topicmaps.org/xtm/1.0/core.xtm#"

association($ASSOC) :- _q($TM, ASSOCIATION, $I, Q, $ASSOC).

association-role($ASSOC, $ROLE) :-
  _q($TM, ASSOCIATION, $I, Q, $ASSOC),
  _q($ASSOC, $TYPE, $ROLE, $SCOPE, $PLAYER),
  _q($TYPE, META_TYPE, $I2, Q, ASSOCIATION_ROLE).

direct-instance-of($INSTANCE, $TYPE) :-
  xtm:class-instance($INSTANCE : xtm:instance, $TYPE : xtm:class).

instance-of($INSTANCE, $TYPE) :-
  xtm:class-instance($INSTANCE : xtm:instance, $DTYPE : xtm:class),
  _self-or-supertype($DTYPE, $TYPE).

occurrence($TOPIC, $OCC) :-
   _q($OTYPE, META_TYPE, $I, Q, OCCURRENCE),
   _q($TOPIC, $OTYPE, $OCC, $S, $V).

reifies($REIFIER, $REIFIED) :- _q($REIFIER, REIFIES, $I, Q, $REIFIED).

resource($OBJ, $URI) :- {
   _q($OTYPE, META_TYPE, $I, Q, OCCURRENCE),
   _q($TOPIC, $OTYPE, $OBJ, $S, $URI) |
   _q($TN, VARIANT, $OBJ, $S, $URI)
}, _is-uri($V).


role-player($ROLE, $TOPIC) :-
  _q($ASSOC, $TYPE, $ROLE, $SCOPE, $PLAYER),
  _q($TYPE, META_TYPE, $I, Q, ASSOCIATION_ROLE).
```

```
scope($OBJ, $TOPIC) :-
  _q($SUBJ, $PROP, $OBJ, $SN, $VAL),
  _q($SN, SCOPE_MEMBER, $I, Q, $TOPIC).

source-locator($OBJ, $URI) :- _q($OBJ, ITEM_IDENTIFIER, $I, Q, $URI).

subject-identifier($TOPIC, $URI) :-
  _q($TOPIC, NODE_URI, $I, Q, $URI),
  not(_q($TOPIC, TYPE_INSTANCE, $I, Q, INFORMATION_RESOURCE)).

subject-locator($TOPIC, $URI) :-
  _q($TOPIC, NODE_URI, $I, Q, $URI),
  _q($TOPIC, TYPE_INSTANCE, $I, Q, INFORMATION_RESOURCE).

topic($TOPIC) :- _q($TM, TOPIC, $I, Q, $TOPIC).

topic-name($TOPIC, $NAME) :-
   _q($NTYPE, META_TYPE, $I, Q, TOPIC_NAME),
   _q($TOPIC, $NTYPE, $NAME, $S, $V).

topicmap($TM) :- _q($TM, TOPIC, $I, Q, $TOPIC).

type($OBJ, $TYPE) :- {
  /* topic name, occurrence, or association role */
  _q($PARENT, $TYPE, $OBJ, $S, $VAL),
  _q($TYPE, META_TYPE, $I2, Q, $METATYPE) |
  /* association */
  ($OBJ, TYPE, $I, $S2, $TYPE)
}.

value($OBJ, $VAL) :- {
   _q($TYPE, META_TYPE, $I, Q, $METATYPE),
   { $METATYPE = TOPIC-NAME | $METATYPE = OCCURRENCE },
   _q($TOPIC, $TYPE, $OBJ, $S, $V) |
   _q($TN, VARIANT, $OBJ, $S, $V)
}, not(_is-uri($V)).

value-like($OBJ, $VAL) :- value($OBJ, $REALVAL), _is-like($REALVAL, $VAL).

variant($TN, $VAR) :- _q($TN, VARIANT, $VAR, $S, $VAL).
```

There is a `base-locator` predicate in tolog which corresponds to the `[base locator]` property of TMDM. This property no longer exists in TMDM, and so it is not defined here.

The `object-id($OBJ, $ID)` predicate produces a unique ID for every topic map object. There are no constraints on the ID beyond that it must be unique within the current topic map, and that it must be a string.

### 3.8 Dynamic predicates

Producing a dynamic association predicate from a topic $t$ requires a supporting predicate that not visible in the language, called `_pair($PAIR, $T1, $T2)`. The arguments to dynamic association predicates are pairs (written `topic1 : topic2` in the syntax), and the predicate is used to get the components of the pair.

For a dynamic association predicate with one parameter, use the following predicate definition:

```
t($P) :-
  _pair($P, $TOPIC, $ROLETYPE),
  role-player($ROLE, $TOPIC),
  type($ROLE, $ROLETYPE),
  association-role($ASSOC, $ROLE),
  type($ASSOC, t).
```

For dynamic association predicates with two parameters, use the following predicate definition:

```
t($P1, $P2) :-
  _pair($P1, $TOPIC1, $ROLETYPE1),
  _pair($P2, $TOPIC2, $ROLETYPE2),
  role-player($ROLE1, $TOPIC1),
  type($ROLE1, $ROLETYPE1),
  association-role($ASSOC, $ROLE1),
  type($ASSOC, t).
  association-role($ASSOC, $ROLE2),
  $ROLE1 /= $ROLE2,
  type($ROLE2, $ROLETYPE2),
  role-player($ROLE2, $TOPIC2),
```

How to extend this to any number of parameters should be obvious.

To produce a dynamic occurrence predicate from the topic $t$, use the following predicate definition:

```
t($T, $V) :- occurrence($T, $O), type($O, t),
             { value($O, $V) | resource($O, $V) }.
```

To produce a dynamic name predicate from the topic $t$, use the following predicate definition:

```
t($T, $V) :- topic-name($T, $N), type($N, t), value($N, $V).
```

## 4 Conclusion and further work

This paper has presented a query algebra for tolog based on the Q model. This gives the query language a formal definition, which gives implementors a much better foundation for producing interoperable implementations. It can also serve as the foundation for work on optimization of tolog queries and type inferencing in tolog queries. Some implementations already support both optimization and type inferencing, but a formal basis is needed to improve both aspects. For this, however, further study of the properties of the query algebra is needed.

Some algebraic properties are immediately obvious. For example, the $\sim$ relation is obviously both reflexive and symmetric, but clearly neither transitive nor total. This implies that the $\oplus$ operator is commutative. It's also clear that $M \oplus M = M$ for all $M \in \mathcal{M}$. $\oplus$ should also be associative, but proving this requires more work. The properties of $\cup$, representing OR, are already known, but more work is required to establish whether $\cup$ is distributive over $\oplus$. The current definition of OPTIONAL is also sub-optimal, in that it presents significant obstacles for optimizations. More work is necessary to determine whether this can be overcome.

More work is also required in order to precisely specify the circumstances under which a tolog query is safe, in the sense that it does not produce infinite results.

Finally, more work is needed in order to establish what the possible sets of values for each variable in a tolog query is, based on the possible sets of values for the parameters to the predicates used in the query. This would make it possible to do type inferencing on a query to tell what types of values a variable may have, which is useful both for optimization and to help programmers find logical errors in their queries.

## References

[Ahmed05] Ahmed, Kal: *Topic Map Relational Query Language – TMRQL.* NetworkedPlanet white paper, 2005. http://www.networkedplanet.com/download/TMRQL.pdf

[Garshol05] Garshol, Lars Marius: *Q: A model for Topic Maps.* Proceedings of Extreme Markup 2005, IDEAlliance, August 1-5, Montréal, Canada. http://www.ontopia.net/topicmaps/materials/quads.html

[Garshol05b] Garshol, Lars Marius: *tolog – Language tutorial.* Ontopia Knowledge Suite documentation, published on Ontopia web site. http://www.ontopia.net/omnigator/docs/query/tutorial.html

[Garshol05c] Garshol, Lars Marius: *The Built-in tolog Predicates – Reference Documentation.* Ontopia Knowledge Suite documentation, published on Ontopia web site. http://www.ontopia.net/topicmaps/materials/tolog-predicate-reference.html

[ISO13250-2] ISO 13250-3: Topic Maps – Data Model; International Organization for Standardization; Geneva. http://www.isotopicmaps.org/sam/sam-model/

[Liu99] Liu, Mengchi. *Deductive database languages: problems and solutions.* ACM Computing Survey 31, 1 (Mar. 1999), 27-62. DOI=http://doi.acm.org/10.1145/311531.311533

[N0492] *TMQL Use Case Solutions*. ISO JTC1/SC34, document N0492, 2004-03-16. http://www.jtc1sc34.org/repository/0492.htm

[Robie01] Robie, Jonathan; Garshol, Lars Marius; Newcomb, Steve; Biezunski, Michel; Fuchs, Matthew; Miller, Libby; Brickley, Dan; Christophides, Vassilis; Karvounarakis, Gregorius. *The syntactic web*. Markup Languages 3, 4 (Sep. 2001), 411-440. DOI= http://dx.doi.org/10.1162/109966202760152176. Available from http://www.w3.org/XML/2002/08/robie.syntacticweb.html

[Seaborne05] Seaborne, Andy; Prud'hommeaux, Eric. *SPARQL Query Language for RDF*. W3C Working Draft 21 July 2005. http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050721/

[Strychowski05] Strychowski, Jakub. *Concept Glossary Manager – Topic Maps Engine and Navigator*. forthcoming, to be published in proceedings of TMRA'05.