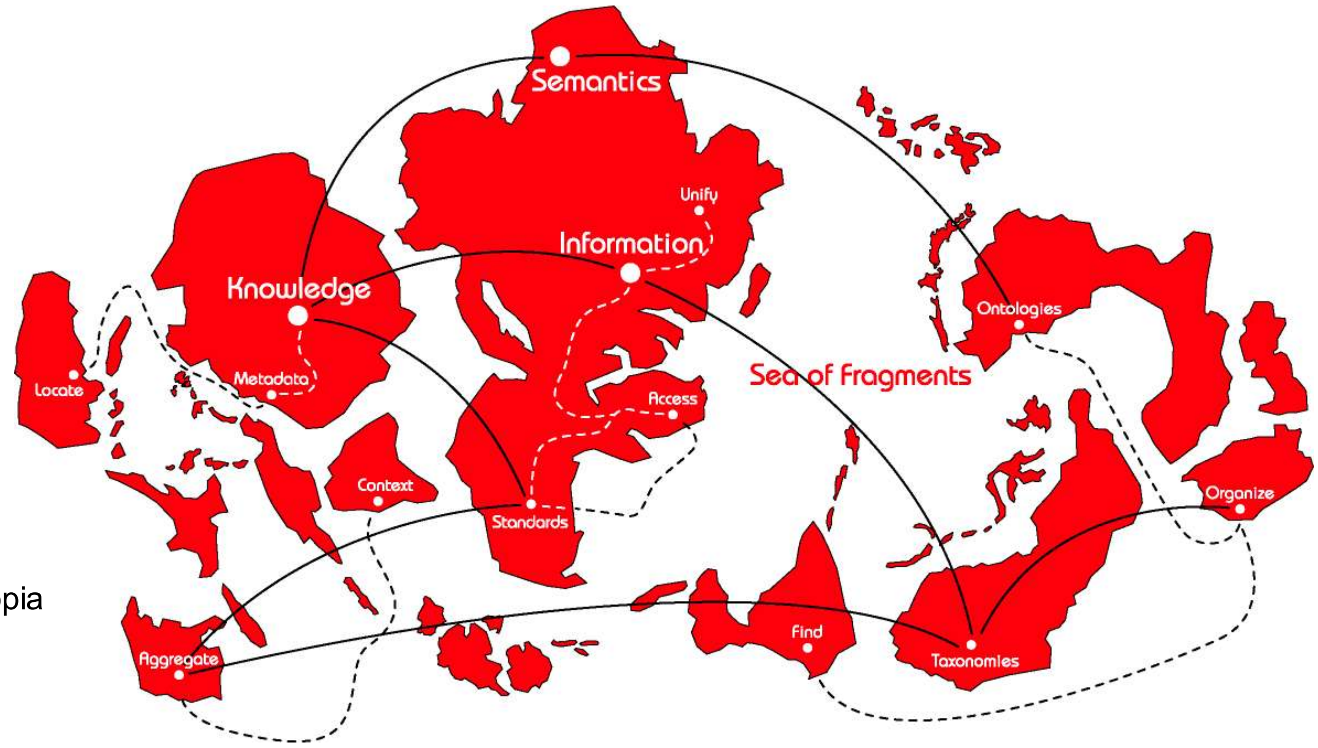
A Query Algebra for tolog

Formalizing tolog



TMRA '05

Lars Marius Garshol
Development Manager, Ontopia
<larsga@ontopia.net>

2005-10-05

Overview

- Quick introduction to tolog
- Superficial intro to the query algebra
- Conclusions and further work

Quick introduction to tolog

tolog in 5 minutes

A brief tolog history

- **tmlog**
  - the original idea came from thinking about using Prolog to query topic maps
  - this resulted in a Jython prototype in December 2000
  - which again turned into a paper for XML Europe 2001 in May 2001
- **tolog 0.1**
  - the first proper version of the language
  - implemented in Java in OKS 1.3 in autumn 2002
  - later also implemented in TM4J
- **tolog 1.0**
  - first version to be able to query all of topic maps
  - adds on and extends 0.1
  - implemented in OKS 2.0, released December 2003
  - currently used as the foundation for many commercial projects
- **The query algebra covers all of tolog 1.0**

Understanding tolog

- tolog does querying by matching a query against the data
- In this process variables are bound to values
- A tolog query result is basically a table with the variables as columns and each set of matches as a row
- Each row represents a set of values that make the query true

A	B
Zandonai, Riccardo	Mascagni, Pietro
Mascagni, Pietro	Ponchielli, Amilcar
Puccini, Giacomo	Ponchielli, Amilcar

Query:
Return all composers who were pupils of another composer, plus the teacher

pupil-of($A : pupil, $B: teacher)?

Building queries

- **AND**
  - born-in($PERSON : person, $PLACE : place),
    located-in($PLACE : containee, italy : container)?
- **OR**
  - **{** premiere($OPERA : opera, $CITY : place) |
    premiere($OPERA : opera, $THEATRE : place),
    located-in($THEATRE : containee, $CITY : container) **}** ?
- **NOT**
  - born-in($PERSON : person, $PLACE : place),
    located-in($PLACE : containee, italy : container),
    not(instance-of($PERSON, composer))?
- **OPTIONAL**
  - instance-of($COMPOSER, composer),
    { date-of-birth($COMPOSER, $DATE) **}** ?

Other tricks

- **Projection**
  - select $PERSON from
    born-in($PERSON : person, $PLACE : place),
    located-in($PLACE : containee, italy : container)?
- **Counting**
  - select $COMPOSER, count($OPERA) from
    composed-by($COMPOSER : composer, $OPERA : opera)?
- **Ordering**
  - instance-of($PERSON, person) order by $PERSON?
- **Paging**
  - instance-of($PERSON, person) order by $PERSON limit 5?
  - instance-of($PERSON, person) order by $PERSON limit 5 offset 5?

Three kinds of predicates

- **Built-in predicates**
  - instance-of, topic-name, role-player, association-role, ...
  - =, /=, <=, ...
- **Dynamic predicates**
  - generated from association, occurrence, and name types
  - born-in, located-in, ...
- **User-defined predicates**
  - inspired-by($X, $Y) :-
    composed-by($X : composer, $OPERA : opera),
    based-on($OPERA : result, $WORK : source),
    written-by($WORK : work, $Y : writer).

The query algebra

A superficial look

A query algebra? What's that?

- **Basically, a set of mathematical operators that correspond to the tolog language constructs**
- **This includes**
  - a mathematical model of Topic Maps,
  - a mathematical model of tolog result sets,
  - a mathematical notion of what predicates are,
  - a set of operators on result sets
- **All of this is effectively a mathematical mirroring of tolog**

Great! So what?

- **The query algebra is a formal definition of what the language *does***
  - this did not exist before
  - now we know what to implement, and other implementors know, too
- **The query algebra is the key to optimizations**
  - query optimization is the art of automatically transforming slow queries into fast queries *that give the same result*
  - the algebra tells us what modifications we can make to a query without changing the results
  - this is similar to how normal algebra says that 5*3 + 5*2 = (2 + 3) * 5
- **The query algebra is the key to type inferencing**
  - when using the built-in predicates developers would often screw up
  - for example, the same variable would be used as a topic name and as a string
  - type inferencing allows us to tell the developer to make his[1] mind up
  - type inferencing is *hard*, and the query algebra tells us how to do it

[1] I've never seen a female developer have this problem

A formal model for Topic Maps

- **In the paper I use the Q model**
  - this was first presented at Extreme Markup earlier this year
- **How Q works**
  - a model instance is a set of quintuples
  - (subject, property, identity, context, value)
  - the first four elements are identifiers, the last can be an identifier or a value
  - the identity of a quint makes it possible to talk about it (yes, reification)
  - the context is the identifier of a set of topics making up a scope
- **The Extreme paper contains**
  - a mapping from any TMDM instance to a Q instance
  - a mapping from Q instances following these conventions to TMDM
  - the same for RDF
  - TMDM-in-Q instances can be treated as RDF
  - RDF-in-Q instances can, once annotated slightly, be treated as topic maps

ontopia

TMDM and Q

Basically, Q tells you how to implement TMDM on a quad store...

The formal model, formally presented

- $\mathcal{I}$ **is the set of all identifiers**
  - an identifier is just an opaque token
  - it doesn't mean anything by itself, it just identifies something
- $\mathcal{L}$ **is the set of all literals**
  - these are data values like strings, integers, URIs, etc
- $\mathcal{A}$ **is the union of** $\mathcal{I}$ **and** $\mathcal{L}$
- **A model is a subset of (**$\mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{A}$**)**
- **Constraints**
  - you can't have two quints in a model with the same id
  - you can't use a quint id as a property
  - you can't use a quint id as a context

tolog query results

- **Matches are sets of (key, value) pairs**
  - the keys are tolog variables
  - the values are values to which the variables are bound
  - duplicate keys cannot occur in the same match
- **Match sets are sets of matches**
  - these correspond to tolog query results

Match set example

- **The expression date-of-birth($PERSON, DATE) would produce a match set like this:**
  - {**{**($PERSON, lmg), ($DATE, 1973-12-25)**}**,
    {($PERSON, stine), ($DATE, 1973-03-24)**}** ... }

Predicate applications

- **Predicates become functions in the query algebra**
  - f(Q, s) – where Q is a topic map, and s is the argument tuple
  - instance-of(Q, ($P, person))
- **The result of a function is always a match set**
  - variables in the argument tuple are bound in the match set
  - filtering by literals is already done

AND

- **e, e' maps to e ⊕ e'**
- **The definition of ⊕ requires another concept**
  - m ~ m' if the matches are compatible
  - that is, if no variables in the two matches contain different values for the same variable
  - M ⊕ M' can now be defined as the set of unions of pairs of matches in M and M' which are compatible
- **Formal definitions**

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

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

An example

- **born-in($P : person, $C : place),
  located-in($C : containee, italy : container)?**
- **The born-in produces all (person, city) combinations where the person is born in the city**
  - e = {**{**($P, lmg), ($C, lærdal)**}**, **{**($P, puccini), ($C, lucca)**}**}
- **The located-in produces all cities in Italy**
  - e' = {**{**($C, lucca)**}**, **{**($C, roma)**}**}
- **The result of e ⊕ e' is**
  - {($P, lmg), ($C, lærdal)} is lost, because e' has no compatible matches
  - {($P, puccini), ($C, lucca)}} is compatible with {($C, lucca)} from e'
  - the last two matches are unioned, which produces
    - {($P, puccini), ($C, lucca)}}
- **Note that if there are no common variables you get a cross-product...**

OR

- **{ e | e' } maps to e ∪ e'**
- **This is straightforward, but there are issues with it**
  - if all matches in e have variable v bound, this doesn't mean those from e' need to
  - the resulting match set can be non-homogenous
  - this needs to be formalized and further described in the algebra

NOT

- **NOT is not trivial...**
  - essentially, what is done is to produce all possible combinations of the variables used in the NOT, then subtract those matched by the negated expression
- **not(e) thus maps to**

$$\Pi(\beta(\mathcal{A}^{|V'|}, V) - e, V \cap V')$$

Built-in predicates

- **The built-in predicates are all defined in terms of a _q predicate**
- **This predicate operates directly on the Q model instance**
- **For example:**
  - association-role($ASSOC, $ROLE) :-
    _q($TM, ASSOCIATION, $I, Q, $ASSOC),
    _q($ASSOC, $TYPE, $ROLE, $SCOPE, $PLAYER),
    _q($TYPE, META_TYPE, $I2, Q, ASSOCIATION_ROLE).
- **Dynamic predicates are mapped to built-in predicates**

The _q predicate

- **The definition of the _q predicate is very simple**
  - $q(Q, p) = \beta(Q, p)$
- **The β function can take a set of tuples, and match it against a tuple of variables and literals**
  - the tuple set is filtered against the literals, and then
  - matches with bindings for the variables are produced
- **This makes defining _q trivial**

Finishing up

What's done, and

What's not

What about TMQL?

- **tolog is the foundation of the OKS at the moment**
  - TMQL won't be here for a while yet
  - meanwhile we needed a proper definition of tolog
- **This work is useful input to TMQL**
  - I've now learned to create a query algebra without getting in anyone's way
  - we now have an alternative query algebra to judge the TMQL one against
- **Ontopia wants to support TMQL**
  - having query algebras for both tolog and TMQL makes it easier to see how to do that
    - can TMQL be compiled to tolog?
    - can tolog be compiled to TMQL?
    - is there a common subset?

Conclusion

- **The query algebra is done (mostly)**
- **The algebraic properties are only partly known**
  - proving them is doable, but takes a little work
- **The type inferencing is not done**
  - again, it's doable, but takes a little work