TMQL language proposal – apart from Path Language

Stockholm, March 2010

Feedback from the committee appears like this

Scope

- What else TMQL offers apart from the Path Language?

and

- How the Path Language is integrated with the rest of TMQL?

  - Does the Path Language have a life of its own?
  - Predicate Invocation
  - SELECT
    - Binding variables
    - Left and right joins?
    - Boolean expressions and exists clauses
  - FLWR, XML content and Topic Map content
  - Auto-atomification
  - Tuples and Tuples Sequences
  - Query context
  - Environment

Does the Path Language have a life of its own

- Should it be possible to use path expressions on their own?
  - Should we have variables then? How do we bind them?
    - $p / email
  - What does the path expression return?
- Alternatively, we can use path expressions with variables within the higher TMQL levels (SELECT/FLWR).
- We can also use path expressions within other frameworks – template systems for example (where variables are bound by the application).

Yes, path expressions can be used on their own
Variables as parameters are allowed
Perhaps parameters should have another syntax

Predicate Invocation

- All agree that the Tolog-like predicate invocation should be supported.
- Should the predicate syntax be integrated with the Path Language syntax?
  - The predicate:

```
association-type(role1: player1,
                 role2: player2,
                 role3: player3)
```

Predicate Invocation (continue)

– As part of the path expression with input and output:

```
player1 / association-type(role1 -> role2,
                           role3: player3) = player2
```

This is the normal association traversal operator, with an additional filter matching the player of role3

Predicate Invocation (continue)

Path expression without input:

```
association-type( -> role2,
                  role1: player1,
                  role3: player3)
```

Matches all associations of the given type that also have the correct players for role1 and role3 (which are role *types*), and then returns the player of the association role of type role2

Kind of like
association-type(r1 : p1, r3 : p3, r2 : $output)

Predicate Invocation (continue)

– Path expression without output:

```
player1 / association-type( role1 ->,
                            role2: player2,
                            role3: player3)
```

This doesn't return anything. To make it do something useful you must use $variables, which are then bound as in predicates.

SELECT

```
select [ distinct ] < value-expression >
  [ from value-expression ]
  [ where boolean-expression ]
  [ order by < value-expression > ]
  [ offset value-expression ]
  [ limit value-expression ]
```

- The **value-expression**s are Path Language expressions.

SELECT – Binding Variables (#1)

- The Tolog/Toma way:

```
select $topic, $topic / name::
  where $topic / type = company
```

Variables get all the possible values combinations that make the WHERE clause true.

In Tolog the above WHERE clause would be written as:

```
{ instance-of($TOPIC, company) |
  type($TOPIC, company }
```

The committee likes binding variables in path expressions, *if we can make it work.*

SELECT – Binding Variables (#2)

But, the path expression language allows us to write:

```
lmg / email
```

Which means that we take all the names, occurrences or roles of type **`email`**. That means that:

```
lmg / $t
```

will bind **`$t`** to all the types of the names, occurrences or roles of **`lmg`**.

And the following is not very clear:

```
$t1 / $t2
```

From the Tolog point of view, the problem is that the path expression language allows us to write “anonymous” predicates:

```
any-predicate-taking-two-inputs($t1, $t2)
```

SELECT – Binding Variables (#3)

This can be solved in the following ways:

- Do nothing – the users will avoid this as they will not understand what it gives.
- When the variable is not constraint to certain type, assume topic as default type to a variable (not that elegant, as variables are allowed to mix values of different types).
- When the variable type is not constraint – generate an error.

SELECT – Binding Variables (#4)

– We could forbid having anonymous predicates in the path expression. So:

```
lmg / email
```

Will have to be written as:

```
lmg / occurrence::email
```

– Other option could be to only forbid having variables with anonymous predicates:

So:

```
lmg / $t
```

Will have to be written as:

```
lmg / occurrence::$t
```

We choose either this one, or the one on the next slide.

SELECT – Binding Variables (#5)

– Avoid the whole problem by allowing variables to get their value only by assignment:

```
select $topic, $n
 where $topic = / company and
       $n = $topic / name::
```

Binding Variables – another example (#1)

```
select $p where
  employed-by(employer: bouvet, employee: $p) and
  lives-in(location: oslo, located: $p)
```

Binding Variables – another example (#2)

Integration of path expressions:

The order of the expressions within the where clause should not affect the result of the query

```
select $p where
  employed-by(employer: bouvet, employee: $p) and
  $p / lives-in(located -> location) = oslo

select $p where
  $p / lives-in(located -> location) = oslo and
  employed-by(employer: bouvet, employee: $p)
```

Binding Variables – another example (#2)

Integration of path expressions:

```
select $p where
  $p / lives-in(located -> location) = oslo and
  $p / employed-by(employee -> employer) = bouvet
```

If we don't allow binding of variables in path expressions this won't work

SELECT – The JOINS Problem

- How to get all the names of all companies in the topic map, and for each name present its variants if there is at least one such variant or null if there is no such variant.
  - In Tolog:

```
select $topic, $n, $v from
    instance-of($topic, company),
    { topic-name($topic, $n),
      variant($n, $v) }?
```

SELECT – The JOINS Problem (continue)

- Alternative – OPTIONAL keyword:

```
select $topic, $n, $v
  where $topic = / company and
  $n = $topic / name:: and
  optional $v = $n / variant::
```

- Another alternative – XOR keyword:

```
select $topic, $n, $v
where $topic = / company and
      $n = $topic / name:: and
      ($v = $n / variant:: XOR $v = null)
```

- Yet another alternative – binding variables in the Path Language filters:

```
select $topic, $topic / name:: [.=$n], $n / variant::
 where exists $topic = / company
```

SELECT – The JOINS Problem (continue #2)

- And another: bind in the query

```
select $topic, $n, $n / variant::
 where $topic = / company and
       $n = $topic / name::
```

- With the XOR inside the path expression

```
select $topic, $n, $v
  where $topic = / company and
  $n = $topic / name:: and
  $v = ($n / variant:: || null)
```

We like the last one. However, we may still need OPTIONAL and XOR.

Where Variables Are Bound

- Where variables can be bound?
  - Only in the WHERE clause or also in the SELECT clause?

```
select $topic / name:: @$scope, $scope
  where $topic / name:: = 'lung'
```

```
$topic / name:: | $scope
----------------+----------
           lung | english
           long | dutch
```

Some people like this.

Others don't like binding variables and producing new rows in the select part.

Where Variables Are Bound

- Where variables can be bound?
  - Only in the WHERE clause or also in the SELECT clause?

```
select $n, $n / scope::
where $topic / name:: = 'lung' and
      $n = $topic / name::
```

This how you would have to write the query if we don't allow new rows and variable binding in select.

```
                $n | $scope
   ----------------+----------
              lung | english
              long | dutch
```

Path expressions in the select always produce *exactly one* value. If the output set is empty, that value is null. If it has more than one value, one is picked at random.

Where Variables Are Bound

- Where variables can be bound?
  - Only in the WHERE clause or also in the SELECT clause?

```
select $n, $scope::
where / topic:: [ name:: = 'lung']
      / name:: [ . = $n ] / scope:: [ . = $scope ]
```

```
                $n | $scope
-----------------+----------
              lung | english
              long | dutch
```

It's also possible to do this entirely in the path language, if we allow variable binding

Where Variables Are Bound

- Where variables can be bound?
  - Only in the WHERE clause or also in the SELECT clause?

```
select $n, $scope::
where `lung` / has-value::
                    [ parent:: / name:: = $n ]
      / scope:: [ . = $scope ]
```

We could introduce more axes to allow a different style of query.

```
                $n | $scope
  ----------------+----------
              lung | english
              long | dutch
```

Possible shorthand syntax for assignment

- Proposal to make this look a bit nicer

```
select $n, $scope::
where / topic:: [ name:: = 'lung']
      / name:: as $n / scope:: as $scope
```

We observe that this removed
an awful lot of noise.

Clarification

select $c, $p where

$c = / company and $p = / person

Assuming TM has 5 companies and 3 people, the result would be 15 rows

Existential semantics

- What about EXISTS clause?
  - Should EXISTS be included for completeness?
    - (EXISTS is currently always implied)
  - Should SOME, AT LEAST and AT MOST be supported?
- What about FORALL clause?

We don’t want any of these new operators.

FLWR, XML Content and Topic Map Content

```
[ for binding-set ]
[ where boolean-expression ]
[ order by < value-expression > ]
 return content
```

- In Seattle, Graham and Rani thought that FLWR can be left out for now, together with the XML content and Topic Map content, and added later (Benjamin mentioned that it seems easy to add). Should we reconsider?
  - FLWR do not suffer from the JOIN problems of the SELECT.
  - FLWR seems to be a better syntax when dealing with graphs.
  - XML content and Topic Map content can be created easily only with FLWR.

We really don't want to do this in the first version of TMQL. We need to finish this thing!

Auto-atomification

- The Path Language defines type conversion functions.
- In many situations the type is known (e.g. comparison to string, function calls etc.)
- In few cases (in the SELECT clause for example), a default type can be set as part of the query context.
  - select $p / name:: where $p / type:: = person
  - by default this returns the name object
- The proposal is to have something like this
  - %pragma output-type string
  - select $p / name:: where $p / type:: = person
  - this now returns the string value

We don't want this.

Tuples and Tuples Sequences

- In Seattle it was agreed that the Tuples and Tuples Sequences section (section 4.8) in the draft is good.
- Do we still think so?

Yes.

Query Context

- Variables:
  - Should we have anonymous variables?
    - E.g: $_foo, or $_, ...
  - Should we have built-in special variables?
  - Should we have primes?
    - E.g: $A and $A’ (with the implicit rule that $A != $A’)

Anonymous: we think we don’t need it, given *.

Built-in: absolutely not.

Primes: absolutely not.

Environment

- Directives
- Pragmas
  - Taxonometry: is this still needed?

What about isa?

Do we need the **isa** operator:

```
select $p where
   $p isa person
```

We like isa. It's transitive.

Or the **type** axis and the **type-instance** predicate are sufficient:

```
select $p where
  $p / type:: = person

select $p where
  tmdm:type-instance(type: person, instance : $p)
```

Using isa in path expressions

/ topic:: [. isa person]

Possible interpretations

(a isa b) => (a / type:: = b)

or

(a isa b) => (a [. / type:: = b])

The second makes it possible to write

/ topic:: isa person / name::

Otherwise you'd have to write

/ topic:: [. isa person] / name::

We already have:

@foo => [. / scope:: = foo]

We're not concluding on this right now. Rani will consider it further.

We will also consider ako.

Scope as first-class object

- This is not possible now. Should it be?

```
                $n | $scope
-------------------+----------
              lung | english
              long | dutch
             lunge | { norwegian, foo }
```

Maybe.