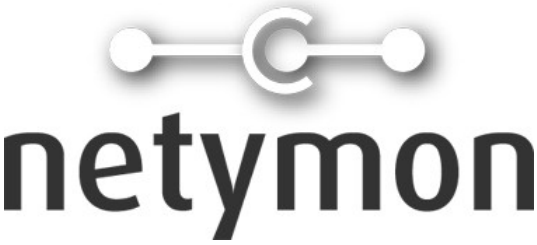

Kowari Documentation

ITQL COMPOUND SYNTAX EXTENSION

netymon pty ltd

Unit 3/13 Denham St Annerley 4103 Australia - email: mail@netymon.com – acn: 113 088 145

Motivation

One of the strengths of RDF is its ability to degrade gracefully in the presence of ambiguity or unknowns. One way this is achieved is by the extensive use of blank nodes. Logically these correspond with skolemised existential quantification, and consequently permit the discription of entities that cannot yet be identified. One unfortunate side-effect of this approach is to substantially increase the number of constraints required to query existentially defined entities. With the introduction of the kowari resolver spi (and especially with the new transformation api), we have increased our ability to model different data sources directly as rdf. Specifically the ability to model non-rdf sources as rdf leads to a signifigant increase in the number of queries complicated by the need to resolve existential quantification.

Summary of New Syntax

The current solution is to provide syntactic sugar within the itql interpreter to allow the collapsing of common existential query patterns into single compound constraints. The new syntax has been derived directly from N3 and SPARQL, and is defined as follows:

```
[ <p1> <o1> : <p2> <o1> , <o2> in <m> ] =>
    $_1 <p1> <o1> in <m> and
    $_1 <p2> <o1> in <m> and
    $_1 <p2> <o2> in <m>
and
  { <s> <p1> <o1> : <p2> <o1> , <o2> in <m> }=>
    <s> <p1> <o1> in <m> and
    <s> <p2> <o1> in <m> and
    <s> <p2> <o2> in <m>
```

the key to the first expansion being the guarentee that $_t is unused and unreferencable by other constraints in the query. We achive this by using the variable $av__N for some arbitary but unique number N.

Example

The proposal that preceded implementation of this feature included the simple addition graph displayed below. This is adequate to provide a concrete example. An elaboration of this graph is used in four of the unit tests.

In		*-i>*	*Out*
1	1	-a>	2
1	2	-b>	3
2	1	-c>	3
2	2	-d>	4
...	...	...	...

This is mapped to rdf as per the following table:

:_a	<add:lhs>	“1”
:_a	<add:rhs>	“1”
:_a	<add:sum>	“2”
:_b	<add:lhs>	“1”
:_b	<add:rhs>	“2”
:_b	<add:sum>	“3”
:_c	<add:lhs>	“2”
:_c	<add:rhs>	“1”
:_c	<add:sum>	“3”
:_d	<add:lhs>	“2”
:_d	<add:rhs>	“2”
:_d	<add:sum>	“4”
...	...	...

A basic query against this model is below. In order to demonstrate the difficulties inherent in the querying of eq-based models we are going to assume that we obtain the values for lhs and rhs from another model in

kowari <rmi://localhost/server1#query>, and that the above arithmetic model is available as a resolver at <rmi://localhost/server1#arith>.

```
select $sum from <rmi://localhost/server1#query>
where
  <doc:12345> <pub:timestamp> $lhs and
  <law:copyright> <law:term> $rhs and
    $_bn <add:lhs> $lhs in <rmi://localhost/server1#arith> and
    $_bn <add:rhs> $rhs in <rmi://localhost/server1#arith> and
    $_bn <add:sum> $sum in <rmi://localhost/server1#arith> ;
```

This query is relatively simple, as the non-rdf data being modeled is only a 3-tuple and we are only interested in performing a single constraint. Yet this complexity increases substantially as we increase the complexity of the model, or nested levels of quantification (example below).

Using the new syntax this query can be written as:

```
select $sum from <rmi://localhost/server1#query>
where
  <doc:12345> <pub:timestamp> $lhs and
  <law:copyright> <law:term> $rhs and
  [ <add:lhs> $lhs : <add:rhs> $rhs : <add:sum> $sum
        <rmi://localhost/server1#arith> ] ;
```

In which the constraint that conceptually will be matched against the 3-tuple of the underlying data is in fact represented syntactically as a 3-tuple in the query.

Unit Test

The extension is tested in the jxunit tests as compound_queries/. It uses a more complicated model then the simple arith model above. Specifically it uses nested existential quantification to differentiate between the concept of 3 and the symbol “3” used to represent the concept. Relationally this is equivalent to normalising the original relation in the example above creating a seperate relation to associate a surrogate key with each number with a string representing the number, and likewise the concept of addition (and in this case multiplication) from the symbol used to represent it. Extracts of those tables are provided below:

PK	*type*	*Symbol*
_1	number	'1'
_2	number	'2'
_3	operator	'*'
...	...	...

PK	*type*	*leftOperand*	*rightOperand*	*operation*	*result*
_4	equiv	_1	_1	_3	_1
_5	equiv	_1	_2	_3	_2
_6	equiv	_2	_1	_3	_2
...	...	...	...	...	...

Note: A strict normalisation would be three relations with a seperate relation to manage the mapping of PK to type.

When mapped to rdf this takes on the following form:

```
<rdf:Description rdf:nodeID="_mul">
  <math:type rdf:resource="math:operator"/>
  <math:symbol>*</math:symbol>
</rdf:Description>
```

```
    <rdf:Description rdf:nodeID="_1">
        <math:type rdf:resource="math:number"/>
        <math:symbol>1</math:symbol>
    </rdf:Description>
...
```

and

```
    <rdf:Description>
      <math:type rdf:resource="math:equiv"/>
      <math:leftoperand rdf:nodeID="_1"/>
      <math:rightoperand rdf:nodeID="_1"/>
      <math:operator rdf:nodeID="_mul"/>
      <math:result rdf:nodeID="_1"/>
    </rdf:Description>
    <rdf:Description>
      <math:type rdf:resource="math:equiv"/>
      <math:leftoperand rdf:nodeID="_1"/>
      <math:rightoperand rdf:nodeID="_2"/>
      <math:operator rdf:nodeID="_mul"/>
      <math:result rdf:nodeID="_2"/>
    </rdf:Description>
...
```

The resulting query to obtain an answer to “3 x 4 = ?” in the original iTql syntax is below:

```
select $result from <rmi://localhost/server1#foobar> where
$lhs <math:type> <math:number> in <rmi://localhost/server1#arith> and
$lhs <math:symbol> '3' in <rmi://localhost/server1#arith> and
$rhs <math:type> <math:number> in <rmi://localhost/server1#arith> and
$rhs <math:symbol> '4' in <rmi://localhost/server1#arith> and
$opr <math:type> <math:operator> in <rmi://localhost/server1#arith> and
$opr <math:symbol> '*' in <rmi://localhost/server1#arith> and
$res <math:type> <math:number> in <rmi://localhost/server1#arith> and
$res <math:symbol> $result in <rmi://localhost/server1#arith> and
$bn_0_0 <math:type> <math:equiv> in <rmi://localhost/server1#arith> and
$bn_0_0 <math:leftoperand> $lhs in <rmi://localhost/server1#arith> and
$bn_0_0 <math:rightoperand> $rhs in <rmi://localhost/server1#arith> and
$bn_0_0 <math:operator> $opr in <rmi://localhost/server1#arith> and
$bn_0_0 <math:result> $res in <rmi://localhost/server1#arith> ;
```

This can be simplified using the extended syntax, resulting in one compound constraint per existential variable to be bound:

```
select $result from <rmi://localhost/server1#compound> where
  { $lhs <math:type> <math:number> :
    <math:symbol> '3' in <rmi://localhost/server1#arith> } and
  { $rhs <math:type> <math:number> :
```

```
    <math:symbol> '4' in <rmi://localhost/server1#arith> } and
  { $opr <math:type> <math:operator> :
    <math:symbol> '*' in <rmi://localhost/server1#arith> } and
  { $res <math:type> <math:number> :
    <math:symbol> $result in <rmi://localhost/server1#arith> } and
  [ <math:type> <math:equiv> :
    <math:leftoperand> $lhs :
    <math:rightoperand> $rhs :
    <math:operator> $opr :
    <math:result> $res in <rmi://localhost/server1#arith> ] ;
```

The result is of course “12”, and this query is one of the primary unit tests testing this feature.

Implementation Comments

As this extension is strictly syntactic sugar, this feature is implemented entirely within the parser itself. The key changes include new productions in the grammar:

```
constraint_factor =
  {constraint} exclude? constraint                          |
  {compound} lbrace [subject]:element exists_expression in_clause? rbrace |
  {existential} lbracket exists_expression in_clause? rbracket |
  {expression} exclude? lpar constraint_expression rpar |
  {transitive} transitive_clause |
  {walk} walk_clause ;

exists_expression =
  {eterm} exists_pterm  |
  {colon} exists_expression colon exists_pterm ;

exists_pterm =
  {pterm} [predicate]:element exists_oterm ;

exists_oterm =
  {eo1} oterm |
  {eo2} exists_oterm comma oterm ;

oterm =
  {oterm} [object]:element ;
```

These are linked to methods on the ConstraintExpressionBuilder:

src/jar/itql/java/org/kowari/itql/ConstraintExpressionBuilder.java

```
public class ConstraintExpressionBuilder extends AnalysisAdapter {
    public void caseAExistentialConstraintFactor(
      AExistentialConstraintFactor rawFactor) { ... }
    public void caseACompoundConstraintFactor(
      ACompoundConstraintFactor rawFactor) { ... }
  public ConstraintExpression buildExistential(ConstraintElement subject,
      PExistsExpression rawTerm, AInClause in) { ... }
}
```

These methods in turn make use of a new public method on ItqlInterpreter, nextAnonVariable() which handles the allocation of anonymous variables to support the [ ] syntax. They also delgate to two new classes, CompoundPredListBuilder and ObjectListBuilder to collect their

respective lists of elements. One shortcut used by the existing parser code to handle in clauses in normal constraints is to upcast the in_clause element representing the production from a PInClause to an AinClause, as only the latter provides access to the internal structure of the production. The buildExistential() method uses the same kludge to avoid having to define a seperate builder to collect the element from the in_clause. In a very real sense this is encapsulation gone mad, and is only one of several reasons why sablecc is not recommended for any future parser work in Kowari.