An Executable Semantics of Graph Query Language

12th LDBC TUC Meeting, Amsterdam

Jan Posiadała, Nodes and Edges
Three in one

Executable
Formal
Readable
Executable Semantics
Natural Language Specification
Reference Implementation
Denotational Semantics
Proof of concept: Cypher.PL

• an executable semantics
• of a declarative query language (Cypher)
• in the formal declarative language of logic (Prolog)
• as close to the semantics as possible
• as far from the implementation issues as possible
• a tool for collaborative language design
• and also an artifact of the design process
Why in Prolog?

- fully formalized declarative language
- built-in unification covers Cypher’s pattern matching
- super-native representation of data with terms
- collects multiple matches (evident ambiguity)
- easy constraint verification
- native support for parsing (DCG)
- built-in meta-programming
- **ISO standard** (ISO/IEC 13211 by JTC 1/SC 22/WG 17)
Current source code status

Sociaal-Wetenschappelijke Informatica-Prolog (7.4.1)

Specifies semantics of Cypher 9+
• 1500 lines of code cover openCypher TCK test set
• reflects semantics ambiguity due to driving table order
• extra graphlet values support
  ▪ binary and aggregation functions for union, intersection operations

```cypher
create (a)-[:T]->(b)-[:T]->(c), (b)-[:T]->(d)
match p=()-->()
return agunion(p) as ug, agintersection(g) as ig
```
Cypher.PL basics

I. Abstract query

II. Evaluation environment: Table × Graph

III. Query evaluation
Abstract query representation

\[ \text{match } p1=(a:A), p2=(b:B) \]

Syntax-sugar-free query
Base for semantics definition

**Machine-oriented**
- Verbose
- Explicit
- Unambiguous

**Planner-friendly**
- Minimal ordering constraints
- Unique variable names
- Human-friendly

Mainly for debugging, not a primary goal
Abstract query definition

single_query(single_query(Clauses)) :- maplist(clause, Clauses).

%%%%% Clauses %%%%%
clause(clause(Clause)) :- match(Clause).
%...
clause(clause(Clause)) :- with(Clause),
clause(clause(Clause)) :- return(Clause).

%%%%% Match %%%%%
match(Clause) :-
Clause = match(MatchModifier, Pattern, Where),
match_modifier(MatchModifier),
pattern(Pattern),
where(Where).

match_modifier(no_modifier).
match_modifier(optional).
%match_modifier(mandatory). %in the future

relationship_pattern(relationshipPattern(Variable
,Direction
,RelationshipTypes
,RelationshipRange
,Properties))

relationship_types(relationshipTypes(Types)) :- maplist(name,Types).

relationship_range(relationshipRange(one_one)).
relationship_range(relationshipRange(L,U)) :-
(L=unlimited; integer(L), %L >= 0,
(U=unlimited; integer(U) %U >= 0)

properties(properties(mapLiteral(Properties))) :- maplist(
[(propertyName(PropertyName),Expression)]
>>
(schema_name(PropertyName),expression(Expression),
Properties).

where(where(Expression) :- expression(Expression).
where(no_where).
Evaluation environment: Graph definition

%read (matching) "api"
node(NodeId,Graph) :- member(node(NodeId),Graph).
relationship(NodeStartId,RelationId,NodeEndId,Graph) :- member(relationship(NodeStartId,RelationId,NodeEndId),Graph).
property(NorRId,Key,Value,Graph) :- member(property(NorRId,Key,Value),Graph).
label(NodeId,Label,Graph) :- member(label(NodeId,Label),Graph).
type(RelationshipId,Type,Graph) :- member(type(RelationshipId,Type),Graph).

%write "api"
create_node(NodeId,Graph,ResultGraph) :- ....
delete_node(NodeId,Graph,ResultGraph) :- ....

set_label(NodeId,LabelName,Graph,ResultGraph) :- ....
remove_label(NodeId,LabelName,Graph,ResultGraph) :- ....

create_relationship(NodeStartId,NodeEndId,RelationshipType,RelationshipId,Graph,ResultGraph) :- ....
delete_relationship(RelationshipId,Graph,ResultGraph) :- ....

set_property(Id,Key,Value,Graph,ResultGraph) :- ....
remove_property(Id,Key,Graph,ResultGraph) :- ....
Evaluation environment: Table

create p=(a:A {int: 1, string: 'S', boolean: true, float: 1.1 })-[r:RELTYPE]->(b)
return p, [a,r] as l, {b : a.boolean, f : a.float, n : a.n} as m, gunion(p,p) as g
Evaluation environment: Table definition

```
is_row_of_table(TableRow) :-
is_list(TableRow),
maplist([bind(Name, Value)]>>(name(Name), cypher_value(Value)), TableRow).

%primitives:
%    cypher_null,
%    cypher_string(Value),
%    cypher_integer(Value),
%    cypher_float(Value),
%    cypher_boolean(Value),
%entities:
%    cypher_node(NodeId),
%    cypher_relationship(RelationshipId),
%    cypher_path(NodesRelationshipsAlternatingList),
%    extended with cypher_graphlet(CypherNodesList, CypherRelationshipsList)
%structures:
%    cypher_list(CypherValuesList),
%    cypher_map(NamesCypherValuesPairsList)
```
Query evaluation

\[
eval_query(G, \text{query(Clauses)}, \text{env}(\text{ResultTable}, \text{ResultGraph})) \\
:= \\
\text{foldl}(\text{eval_clause}, \text{Clauses}, \text{env}([], G), \text{env}(\text{ResultTable}, \text{ResultGraph})).
\]

\[
eval\_clause(\text{clause(MatchClause)}, \text{env}(\text{Table}, G), \text{env}(\text{ResultTable}, G)) \\
:= \\
\text{MatchClause} = \text{match(MatchModifier, Pattern, Where)}, \\
\text{findall}(\text{MatchTableRow}, \\
\quad ( \\
\quad \text{member(TableRow, Table),} \\
\quad \text{eval}\_match(MatchModifier, Pattern, Where, G, TableRow, MatchTableRow) \\
\quad ), \\
\quad \text{MatchResultTable}).
\]
Query evaluation: ambiguity handling

```sql
create ({num : 1}),({num : 2})
with *
match (n)
return (collect(n)[1]).num as nn
```

<table>
<thead>
<tr>
<th></th>
<th>nn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Query evaluation: ambiguity handling

eval_clause(clause(Clause), env(Table, Graph), env(ResultTable, ResultGraph)) :-
scall(set(Table), eval_clause_(Clause, Graph), environment_eq,
    env(ResultTable, ResultGraph)).

scall(set(Set), Goal, EqGoal, Result) :-
findall(PermutationResult,
    (permutation(Set, Permutation),
    call(Goal, Permutation, PermutationResult),
    PermutationResults),
gr_by(EqGoal, PermutationResults, ResultsGroups),
member([Result|_], ResultsGroups).

eval_clause_(Clause, Graph, Table, env(ResultTable, ResultGraph)) :-
% particular clause semantics definition

environment_eq(env(Table1, Graph1), env(Table2, Graph2)) :-
permutation(Table1, Table2), isomorphic(Graph1, Graph2).
Intuitiveness of **MERGE**

That makes me wonder if it might be (conceptually possible) to express **MERGE** using subqueries like this:

```sql
MATCH (a)
MATCH {
  MATCH (a)-[:X]->(m {prop: n.prop})
  RETURN n, m
}
OTHERWISE /* query-level xor that has been discussed in the past */
MATCH {
  CREATE (a)-[:X]->(m {prop: n.prop})
  RETURN n, m
}
```

This shows where the problem is: it still would create duplicates and the only way to avoid it I could see would be a graph-level squashing operation of similarly looking entities. And to be the same, giving a real argument why **MERGE** is a core feature.
Intuitiveness of MERGE

eval_merge(merge(patternElement(Variable,Patterns), MergeActions),
    Graph,
    TableRow,
    environment(ResultMatchTable,ResultMergeGraph))
:-
eval_clause(clause(match(no_modifier,pattern([[patternElement(Variable,Patterns)]],no_where)),
    environment([[TableRow],Graph]),
    environment(ResultMatchTable,Graph)),
not(ResultMatchTable = []),
!,
convlist([onMatch(Set),Set]>>true, MergeActions,OnMatchActions),
foldl(eval_merge_actions,
    OnMatchActions,
    environment(ResultMatchTable,Graph),
    environment(ResultMatchTable,ResultMergeGraph)).
Intuitiveness of MERGE

eval_merge(merge(patternElement(Variable,Patterns), MergeActions),
            Graph,
            TableRow,
            environment(ResultCreateTable, ResultMergeGraph))

:-
eval_clause(clause(create(pattern([patternElement(Variable,Patterns)]))),
              environment([TableRow, Graph],
                            environment(ResultCreateTable, ResultCreateGraph)),
              convlist([onCreate(Set), Set]>>true, MergeActions, OnCreateActions),
              foldl(eval_merge_actions,
                   OnCreateActions,
                   environment(ResultCreateTable, ResultCreateGraph),
                   environment(ResultCreateTable, ResultMergeGraph)).
Broad standardization scope to support

- property graph query and update language
- ... including errors definitions, raising and handling
- schema languages for property graphs
- languages interoperability
- session, transaction and concurrency model
- ...
- ...
Proposal

A formal, readable, and executable semantics can and should be both a tool and an artifact of language standardization.
Our latest activity


References


**Formal Semantics of the Language Cypher.**

N. E. Fuchs.

**Specifications are (preferably) executable.**

R. A. Kowalski.

**The relation between logic programming and logic specification.**