2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology

The KGRAM Abstract Machine for Knowledge Graph Querying

Olivier Corby
Edelweiss, INRIA Sophia Antipolis, France
olivier.corby@sophia.inria.fr

Catherine Faron Zucker
I3S, Université de Nice Sophia-Antipolis, CNRS, France
catherine.faron-zucker@unice.fr

***Abstract*—In this paper we present the KGRAM abstract machine dedicated to querying knowledge graphs. It is the result of an abstraction process we performed to reach a generic solution to the problem of querying graphs in various models. We identified high level abstract primitives that constitute the expressions of the query language and the interfaces of KGRAM for both its data structures and its operations.**

***Keywords*-Knowledge graphs, Query languages, Abstract machine, Natural Semantics**

I. Introduction

In this paper we present KGRAM (Knowledge Graph Abstract Machine) dedicated to querying knowledge graphs. KGRAM result from an abstraction process we conducted in order to propose a generic solution to the problem of querying oriented labelled graphs and more specifically knowledge graphs in various models. This work addresses current major challenges related to the multiplication of co-existing knowledge representation languages. With KGRAM we propose to unify reasoning mechanisms for querying knowledge bases in different models. To do so, we identified high level abstract primitives which constitute KGRAM expressions and interfaces. KGRAM can be viewed as an interpreter of a generic query language which manipulates corresponding interfaces for both its data structures and its graph operations.

The abstraction work we performed in the definition of KGRAM has been inspired by the results of the GRIWES [1] project to which we participated. The conception of KGRAM benefited from our experience on the conception and development of the Corese semantic engine[1] dedicated to querying Semantic Web data with internal data structures and operations relying on the Conceptual Graph model [4], [5]. Finally, our idea to build an abstract machine for knowledge graphs relates KGRAM to the YAM abstract machine [7] and the AMaXoS abstract machine [2].

YAM implements the graph programming language GP and enables to perform any operation over oriented labelled graphs. It is a very low level abstract machine whereas we seeked in KGRAM a high level of abstraction: YAM manipulates hashtables whereas KGRAM manipulates node and edge interfaces; the YAM instructions are simple stack operations whereas the KGRAM language is a high level API for querying graphs – which notably generalize and extend SPARQL.

AMaXoS implements the Xcerpt language for querying XML data. In Xcerpt, queries are patterns and answers to queries are instances of patterns. KGRAM and AMaXoS share the same goal of unifying data querying on the Web by subsuming both data representation languages and query languages. However, AMaXoS only considers the data structure while KGRAM takes into account the semantics of nodes and edges in knowledge graphs. Moreover, AMaXoS executes some low level code resulting from the compilation of Xcerpt queries. Like YAM and contrary to KGRAM it is a low level abstract machine.

The genericity of KGRAM is relative to the various graph models it enables to query. Its function for evaluating a query over a graph base only manipulates interfaces — of nodes and edges — and calls for functions of interfaces — of an abstract graph manager, an abstract comparator of node labels and of an abstract constraint evaluator. KGRAM thus enables to query graphs of various models, provided the implementation of these interfaces. This genericity makes KGRAM interoperable in the sense that it enables to exploit graphs coming from different models by connecting different graph managers and constraint evaluators implementing the same interfaces. In the simplest case, KGRAM enables to match oriented labelled graphs by supplying a basic implementation of a comparator of node and edge labels. As further described later in this paper, we developed two implementations of KGRAM interfaces which take into account the semantics of the graphs and then match conceptual graphs with constraints or query RDF graph with an extension of SPARQL.

The KGRAM genericity and interoperability broaden the perspective to distribute the treatment of queries over different knowledge graph bases that may be heterogeneous. This is a multiple perspective. KGRAM can first be viewed as a mean to unify querying over graph-based data in various models. This is a major issue for the Web of Data where RDF/S, Topic Maps, XML or DB data coexist. Moreover combining the results of partial results on different bases should enable the development of mashup applications. Finally, the call of several graph managers in separate parallel threads should enables to tackle with KGRAM the problem of scaling in Web querying.

[1] http://www.inria.fr/sophia/edelweiss/software/corese

978-0-7695-4191-4/10 $26.00 © 2010 IEEE
DOI 10.1109/WI-IAT.2010.144

II. The Abstract Query Language

A. Abstract Syntax

The abstract syntax of KGRAM's query language is given by the following grammar:

```
QUERY  ::= query(NODE *, EXP)
EXP    ::= QUERY | NODE | EDGE | FILTER | PATH
         | and(EXP, EXP) | union(EXP, EXP)
         | option(EXP) | not(EXP) | exist(EXP)
         | graph(NODE, EXP)
NODE   ::= node(label)
EDGE   ::= edge(label, NODE *)
PATH   ::= path(RegExp, NODE, NODE)
FILTER ::= filter(FilterExp)
```

Here is a simple example of an expression which enables to query for authors and titles of documents (the query does not depend on the model of the graphs which are queried).

```
query({node('?x'), node('?title')},
  and(edge('hasCreated',
        {node('?x'), node('?doc')}),
      edge('hasTitle',
        {node('?doc'), node('?title')})))
```

A query is defined by an expression to be evaluated and a list of variables for which the list of values is searched when the query expression is evaluated on the graph which is queried. A QUERY expression enables to express such a query. Its EXP parameter represents the expression to evaluate and its NODE parameters the variables for which the bindings are searched. These variables correspond in a concrete syntax to those of a SELECT clause in an SPARQL-like language or to the parameters of a lambda-expression in the Conceptual Graph model. A QUERY expression also enables to formulate a query nested into another. In that case the result of its evaluation determines bindings for the rest of the evaluation of the embedding query.

NODE and EDGE expressions enable to query for nodes or n-ary relations (hyperarcs) in a hypergraph. The `label` parameter of a NODE or EDGE expression represents the label of a node or an edge in a graph; it is a constant (or a variable for NODE).

The `FilterExp` parameter of a FILTER expression enables to express constraints on the searched nodes in the graph which is queried. It is a boolean expression of a constraint language (interpreted by a filter evaluator given to KGRAM):

```
FilterExp ::= Variable | Constant | Term
Term      ::= Oper(FilterExp *)
Oper      ::= '<' | '<=' | '>=' | '=' | '!='
            | '&' | '|' | '!' | '+' | '-'
            | '*' | '/' | FunctionName
```

Let us note that NODE, EDGE and FILTER expressions are primitive and we will show in section III that they correspond to interfaces of the abstract machine KGRAM.

A PATH expression is a generalization of an EDGE expression. It enables to query for paths of binary relations between two nodes in a graph. An AND (resp. UNION) expression enables to express a conjunction (resp. disjunction) between two expressions. An OPTION expression makes optional the existence of solutions to some expression in the search of solutions to a query. A NOT expression expresses negation as failure. An EXIST expression enables to search for only one solution (the first retrieved). A GRAPH expression enables to specify the knowledge graph upon which the query is evaluated (without such an expression it is a default graph which is considered).

B. Natural Semantics

Natural Semantics has first been introduced by [6] to provide an operational semantics to programming languages. In Natural Semantics the operational semantics of a language is given by a set of inference rules. These rules enable to evaluate the expressions of the language in an environment and produce lists of environments. Therefore the rules of Natural Semantics established for KGRAM's query language describe the evolution of the environment (initially empty) during the evaluation of an expression building up a query.

The following rule 1 governs the way to evaluate an expression for searching an EDGE in a graph. It specifies that the evaluation of such an expression in an environment *ENV* requires to compute the list of environments *LENV* capturing the possible matching of EDGE in the graph which is queried and to merge *ENV* and *LENV*. These two operations are synthesized in the rule bases *match* and *merge* which specify the semantics of the comparator of edge labels and the environment manager of KGRAM (see section III).

$$\frac{match(ENV \vdash EDGE \rightarrow LENV) \wedge \\ merge(ENV, LENV \rightarrow LENV')}{ENV \vdash EDGE \rightarrow LENV'} \quad (1)$$

A similar rule governs the way to evaluate an expression for searching a NODE in a graph.

The following rules 2 and 3 define the way to evaluate a FILTER expression. The rule base *eval* relative to the evaluation of the boolean expression by which a FILTER expression is parameterized exploits the bindings of the query variables embedded in the current environment *ENV*. Rule 2 specifies that if this boolean expression is evaluated to false then an empty environment list (nil) is produced: there is no solution. Rule 3 specifies that otherwise the list produced contains a single element which is the current environment (this list is created with the *list* operator).

$$\frac{eval(ENV \vdash F : false)}{ENV \vdash filter(F) \rightarrow nil} \quad (2)$$

$$\frac{eval(ENV \vdash F : true)}{ENV \vdash filter(F) \rightarrow list\ ENV} \quad (3)$$

For lack of space, we do not detail in this paper the Natural Semantics rules for the other expressions of the KGRAM's language.

C. Some Remarquable Languages

Depending on the subset of expressions that we consider, we define a particular (sub) language. Worth noticing, the NODE and EDGE expressions define a query language corresponding to the one of the Simple Conceptual Graph model [3]. The operationalization of the Natural Semantics rules associated to these expressions corresponds to the search of homomorphisms on labelled graphs whose relations may be n-ary.

The expressions NODE, EDGE, FILTER, AND, UNION, OPTION and GRAPH define a sublanguage which corresponds to the core of SPARQL SELECT-WHERE query pattern extended to n-ary relations. In addition, the EXIST expression corresponds to the ASK query pattern of SPARQL and the notion of nested query captured in the QUERY expression is under review by SPARQL 1.1 WG as well as PATH.

III. The Abstract Machine KGRAM

A. KGRAM's Interfaces

KGRAM accesses the graph through an abstract API that hides the graph's structure and implementation. In other words, KGRAM operates on a graph abstraction by means of abstract structures and functions and it ignores the internal structure of the nodes and edges manipulated in its function of evaluation of a query expression over a target graph. More precisely, the target graph is accessed by node and edge iterators that implement the *Node* and *Edge* interfaces of KGRAM. These are the very same interfaces that operationalize the NODE and EDGE expressions. As a result, KGRAM can process any kind of knowledge graph, in particular conceptual graphs (with n-ary relations) as well as RDF graphs (with binary relations).

KGRAM manipulates not only abstract data structures but also abstract operators:

- KGRAM accesses the target graph through an abtract graph manager which implements its *Producer* interface. This graph manager enumerates the graph nodes and edges (implementing the *Node* and *Edge* APIs) that match the nodes and edges occurring in a given expression (and implementing the same APIs).
- A node and edge matcher implements the KGRAM *Matcher* interface. Depending on the *Matcher* implementation, label comparison consists in testing string label equality or it may take into account class and property subsumption, or compute approximate matching based on semantic similarities, etc.
- Constraints (or filters) are abstract entities that implement the *Filter* interface which corresponds to the FILTER expression. Filters are evaluated by an object that implements the *Evaluator* interface. KGRAM ignores the internal structure of filters, it calls the `eval` function of *Evaluator* on *Filter* objects and passes the *Environment* as argument.

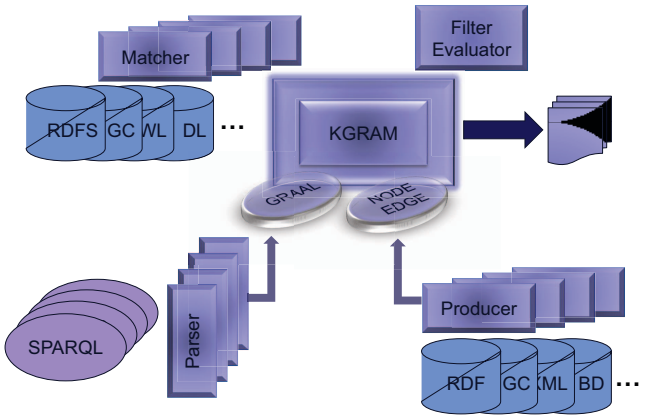

Figure 1. KGRAM in a nutshell

To sum up, KGRAM interprets expressions and implements their natural semantics by using an abstract API. Figure 1 presents the KGRAM's architecture and highlights the high abstraction level we kept while designing KGRAM. The KGRAM query evaluation algorithm uses only abstract interfaces and hence remains independent of any graph implementation and any data structures.

B. KGRAM's Evaluation Function

KGRAM's algorithm for evaluating a query expression implements the natural semantics rules. It specially relies on rule 1 associated to the expression EDGE. The environments produced by these rules represent the (partial) homomorphisms found between the expression an the target graph. KGRAM's algorithm is described below. The `queryStack` argument of the `eval` function represents the stack of expressions participating to the query that is evaluated. Its argument `i` represents the current position in this stack. The function is initially called with the whole query in the stack and a value of zero for i. An instance of KGRAM is created with (1) a `producer` which implements the *Producer* interface, (2) a `matcher` which implements the *Matcher* interface, (3) an `evaluator` which implements the *Evaluator* interface, (4) an environment manager `env` which stores in a stack structure the current environment, i.e. a partial homomorphism described as node bindings and (5) a list of complete homomorphisms (representing the results of the evaluated query).

```
eval(queryStack, i){
  if (queryStack.size() = i){
    store(env); return;}
  exp = queryStack(i);
  switch(exp){
    case EDGE:
```

```
    for (Edge r :
           producer.candidate(exp, env)){
         env.push(exp, r))
         eval(queryStack, i+1);
         env.pop(exp, r);}
    break;
  case FILTER:
    if (evaluator.eval(exp, env))
       eval(queryStack, i+1);
}}
```

In the `switch` control instruction, the blocks labelled by EDGE implement the rule 1 and hence complete the current environment with node and edge bindings between the query and target graphs. The `candidate` function of the graph manager `producer` is called; it takes as argument a NODE or EDGE expression from the stack `queryStack` and the current environment `env`. It uses the environment to retrieve, if any, the nodes in the `exp` expression that are already bound. Therefore it returns the only edges compatible with the bindings in the current environment. These candidate edges are added each one its turn in the current environment as new bindings. The search of a homomorphism eventually succeeds and the partial homomorphism is completed when the summit of the stack is reached: *env* is then added into the result list by calling the function `store()`.

The FILTER block in the `switch` control instruction implements the rules 2 and 3 relative to the expression FILTER. KGRAM then implements the search of homomorphisms under constraints. If the filter evaluates to true, the search of an homomorphism continues with the same environment. Otherwise the partial homomorphism represented by the current environment cannot be completed and a backtrack in the *eval* function enables to go back to a previous level in the stack of expressions `queryStack`, enumerate new candidates and then evaluate the filter in other environments where it may succeed.

The whole natural semantics rule base is implemented in KGRAM by specific blocks integrated to the backbone of the algorithm shown above: each expression has it's own block.

C. KGRAM's Interoperability

We have tested KGRAM's portability by implementing its interfaces *Node*, *Edge*, *Producer*, *Matcher*, *Evaluator* described above with both Corese and Jena. In order to validate KGRAM, we use a RDF base with 25,000 triples and almost 500 queries.

The connection to Corese was almost immediate because KGRAM was designed as an abstraction of the principles of Corese. In this port, KGRAM handles the whole query language and queries RDF graphs implemented as conceptual graphs. We have also ported KGRAM (except property path) on Jena within a Master trainee of Corentin Follenfant. The port requires less than 1000 lines of code and it succeeded. These two implementations testify the genericity of the design of KGRAM and show that the connection to other implementations is easy.

IV. Conclusion and on-going Work

We have presented the KGRAM abstract machine for querying knowledge graphs and its graph based query language. We have established the natural semantics rules for each expressions and these rules represent the specification of the KGRAM algorithm — KGRAM can be viewed as an interpreter of the query language. We have highlighted the high abstraction level of KGRAM and the simplicity of its algorithm which relies on the manipulation of interfaces for both operations and data structures.

We have extended the query language with an XPath extension, similar to its property path, to access XML nodes into some retrieved documents during the processing of an expression.

Finally, our future prospects deals with the distribution of treatments over the Web of Data. We consider to handle this problem by interconnecting different graph managers implementing the KGRAM API, responsible of one knowledge base each. We envision KGRAM as a response element to the problem of scaling in processing the Web of Data and as the keystone to mashup applications combining results of several graph managers.

References

[1] J.F. Baget, O. Corby, R. Dieng-Kuntz, C. Faron-Zucker, F.Gandon, A. Giboin, A. Gutierrez, M. Leclère, M.L. Mugnier, and R. Thomopoulos. GRIWES: Generic Model and Preliminary Specifications for a Graph-Based Knowledge Representation Toolkit. In *Proc. of the 16th International Conference on Conceptual Structures, ICCS 2008, LNCS 5113*, pages 297–310. Springer, 2008.

[2] F. Bry, T. Furche, and B. Linse. AMaXoS Abstract Machine for Xcerpt: Architecture and Principles. In *Proc. of 4th Workshop on Principles and Practice of Semantics Web Reasoning, LNCS 4187*, pages 105–119, 2006.

[3] M. Chein and M.L. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer London Ltd, 2009.

[4] O. Corby, R. Dieng-Kuntz, and C. Faron-Zucker. Querying the Semantic Web with Corese Search Engine. In *Proc. of the 16th Eureopean Conference on Artificial Intelligence, ECAI 2004*, pages 705–709. IOS Press, 2004.

[5] O. Corby and C. Faron-Zucker. Implementation of SPARQL Query Language Based on Graph Homomorphism. In *Proc. of the 15th International Conference on Conceptual Structures, ICCS 2007, LNCS 4604*, pages 472–475. Springer, 2007.

[6] G. Kahn. Natural Semantics. In *Proc. of 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS 87, LNCS 247*, pages 22–39. Springer, 1987.

[7] G. Manning and D. Plump. The York Abstract Machine. *Electron. Notes Theor. Comput. Sci.*, 211:231–240, 2008.