KGRAM: An Abstract Machine to Process Linked Data Graphs

Olivier Corby[1], Catherine Faron Zucker[2], and Fabien Gandon[1]

[1] INRIA Sophia Antipolis-Méditerranée,
2004 route des Lucioles, BP 93, 06902 Sophia Antipolis cedex, France
{olivier.corby, fabien.gandon}@inria.fr http://www-sop.inria.fr/edelweiss/
[2] I3S, Université Nice - Sophia Antipolis, CNRS
930 route des Colles, BP 145, 06930 Sophia Antipolis cedex, France
catherine.faron-zucker@unice.fr

Abstract. We present KGRAM (Knowledge Graph Abstract Machine), an engine to query and mashup linked data. It interprets an abstract language generalizing SPARQL 1.1 to query not only RDF but a larger family of knowledge graph models. KGRAM comes with an API which corresponds to the abstract primitives of this query language and to the abstract graph data structures. The evaluation function of this virtual graph machine only manipulates those abstract structures.

Keywords: Semantic Search, Semantic Mashup, SPARQL 1.1 Interpreter, Graph Abstract Machine, Knowledge Graphs, Linked Data.

1 Introduction

Graph structures are multiplying on the web: social networks, service compositions, clickstreams, thesauri and type hierarchies, frequent patterns, timelines and workflows, communication networks, information flows, etc. These structured metadata form typed graphs: nodes and arcs of these graphs are labeled with types that can support inferences and enrich their use. These typed graphs and the operations they support are our main topic of research. These typed graph not only support logical reasoning but can also be seen as metric spaces to pilot approximate reasoning and querying, or as indexes of knowledge in distributed environments, or as models to make interfaces more intelligible to end-users, or as new frameworks for social structures analysis. We intend to generalize these results to address the growing diversity of graphs on the web.

Here we present the foundations of the open-source platform KGRAM (Knowledge Graph Abstract Machine) dedicated to the querying and mashup of linked data graphs. It results from an abstraction process we conducted to propose a generic solution to the problem of querying oriented labelled multigraphs like RDF data while addressing the multiplication of coexisting knowledge representation languages. It provides unifying reasoning mechanisms for querying various knowledge graph models [5]. The abstract graph model of KGRAM builds upon the results of the GRIWES project to which we participated [1].

KGRAM is designed as the interpreter of an abstract language which generalizes SPARQL 1.1, including aggregate functions, subqueries, negation and property paths. It enables us to query different graph models and even more *any model* — provided that it is capable of producing a graph view of its data —, e.g. XML, relational databases. To achieve this genericity, KGRAM is designed with a strict distinction between the interpreter of the query language and the data graph manager. KGRAM's API reflects both the abstract primitives of the query language and the abstract graph data model. Its interpreter is connected to one or several graph managers implementing KGRAM's interface and its evaluation function only manipulates the abstract graph structures of KGRAM's interface.

This genericity of KGRAM makes it interoperable in the sense that it enables it 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 multigraphs — by supplying a basic implementation of a comparator of node and edge labels. As further described later in this paper, we also developed three other implementations of KGRAM interfaces which take into account the semantics of graphs; one matches conceptual graphs with constraints and the two other ones query RDF graphs, one of them with SPARQL 1.0 and the other with an extension of SPARQL 1.1. KGRAM's genericity and interoperability also allow us to distribute the storage and the processing to scale an application or integrate heterogeneous sources for instance.

In this paper we focus on the theoretical foundations of the abstract graph model and the associated virtual graph machine that specified the implementation of KGRAM. We first present in section 2 KGRAM's abstract graph model. Section 3 is dedicated to KGRAM's abstract query language and its interpreter.In section 4 we present the KGRAM's application programming interface (API) and we show how to use it to build web applications to the querying and mashup of distributed and heterogenous data. We summarize in section 5 KGRAM's capabilities and discuss its limitations.

2 A Graph-based Knowledge Representation Model

Our generic graph-based knowledge representation framework piles up three layers of abstraction [1]: The *structure layer* gathers and defines the basic mathematical structures (e.g. an oriented acyclic labeled graph) that are used to characterize the primitives for knowledge representation. The *knowledge layer* factorizes recurrent knowledge representation primitives (e.g. a fact) that can be shared across specific KR languages. The *language* gathers definitions specific to languages (e.g. an RDF triple).

2.1 Structure Layer

Entity-Relation graphs. Our core representation primitive is intended to describe a set of entities and relationships between these entities; it is called an

Entity-Relation graph (ERGraph). An entity is anything that can be the topic of a conceptual representation. A relationship, or simply relation, might represent a property of an entity or might relate two or more entities. The relations can have any number of arguments including zero and these arguments are totally ordered. In graph theoretical terms, an ERGraph is an oriented hypergraph, whose nodes represent entities and whose hyperarcs represent relations between these entities. A hypergraph has a natural graph representation associated with it: a bipartite graph, with two kinds of nodes representing entities and relations, and edges linking a relation node to the entity nodes arguments of the relation.

Definition 1. *An* ERGraph *is a 4-tuple* $G = (E_G, R_G, n_G, l_G)$ *where:*

- E_G *and* R_G *are two disjoint finite sets respectively, of nodes called entities and of hyperarcs called relations,*
- $n_G : R_G \to {E_G}^*$; $r \mapsto (e_1, ..., e_k)$ *associates to each relation a finite tuple of entities. We note* $n_{G_i}(r) = e_i$ *the* i^{th} *argument of* r*,*
- $l_G : E_G \cup R_G \to L$ *is a labelling function of entities and relations.*

At the structure level, the labels of nodes and relations are just elements of a set L that can be defined in intension or in extension. They get a meaning at the knowledge level. It is sometimes useful to distinguish some entities of a graph. For this purpose we define a second core primitive, called λ-ERGraph.

Definition 2. *A* λ-ERGraph λ_G *is a couple of an ERGraph* G *and a tuple of entities of* G*:* $\lambda_G = ((e_1, \dots e_k), G), e_i \in E_G$*.* k *is the* size *of* λ_G *and* $e_1, \dots e_k$ *are said* distinguished *in* G*.*

Mapping between ERGraphs. Mapping entities of graphs is a fundamental operation for reasoning with ERGraphs.

Definition 3. *Let* G *and* H *be two ERGraphs. An* EMapping *from* H *to* G *is a partial function* $M : E_H \to E_G$*.*

By default an EMapping is partial. This enables us to manipulate and reason on EMappings during the process of mapping graphs. When this process is finished, the EMapping — if any — is total.

We define the proof of a mapping as a kind of reification of the mapping: it provides a static view over the dynamic operation of mapping, enabling us to access information relative to the state of the mapping.

Definition 4. *Let* G *and* H *be two ERGraphs, and* M *an EMapping from* H *to* G*. The EProof of* M *is a set* $M_E = \{(e_H, e_G) \in E_H \times E_G \mid e_G = M(e_H)\}$*.*

We identify specific mappings preserving some characteristics of the graphs. An ERMapping constrains the graph structures being mapped: it is an EMapping that maps each relation in H to a relation in G with the same arity. An EMapping$_{<X>}$ constrains the labelling of entities in the graphs being mapped: it is an EMapping that satisfies a compatibility relation X on entities labels. An ERMapping$_{<X>}$ is both an ERMapping and an EMapping$_{<X>}$. A Homomorphism is a total ERMapping. Proofs are defined for each of these EMappings.

Definition 5. *Let G and H be two ERGraphs, and M an EMapping from H to G. Let H' be the subERGraph of H induced by $M^{-1}(E_G)$. An* ERProof *of M is a couple $P = (M_E, M_R)$ where:*
- M_E is the EProof of M,
- $M_R = \{(r_1, r'_1), \dots (r_k, r'_k)\}$ with $\{r_1, \dots r_k\} = R'_H$ and $\forall i = 1 \dots k, r'_i \in M(r_i)$.

Definition 6. *Let G and H be two ERGraphs, and M an EMapping from H to G. The* EProof_X *of M is a set $M_{EX} = \{(e_1, e'_1, p_1), \dots (e_k, e'_k, p_K)\}$ where:*
- $\{(e_1, e'_1), \dots (e_k, e'_k)\}$ is the EProof of M,
- $\forall i, 1 \leq i \leq k$, p_i is a proof of $(l_G(M(r)), l_H(r)) \in X$.

We make no assumptions on the structure of p_i nor on the means to obtain it. A system for comparing labels should be able to produce such proofs.

Definition 7. *Let G and H be two ERGraphs, and M an EMapping from H to G. An* $\text{ERProof}_{<X>}$ *of M is a couple $P = (M_{EX}, M_{RX})$ where M_{EX} is the $\text{EProof}_{<X>}$ of M and $M_{RX} = \{(r_1, r'_1, p_1), \dots (r_k, r'_k, p_K)\}$ where:*
- $\{(r_1, r'_1), \dots (r_k, r'_k)\}$ is the second element of an ERProof of M,
- $\forall i, 1 \leq i \leq k$, p_i is a proof of $(l_G(M(r)), l_H(r)) \in X$.

The proof of a homomorphism is an $\text{ERProof}_{<X>}$.

Constraints System for Mappings. An EMapping constraint system is a function $\mathcal{C}$ that sets additional conditions that an EMapping must satisfy in order to be correct.

Definition 8. *A* constraint system *for an EMapping M from H to G is a function $\mathcal{C}$ which applies to the triple $E = (H, P, V)$ called* environment*, with P the proof of M and V a binary relation associating to variables v_i a unique entity or relation of H. $\mathcal{C}(E) \in \{true, false, unknown, error\}$.*

An EMapping M satisfies (resp. violates) a constraint system $\mathcal{C}$ if $\mathcal{C}$ evaluates to $true$ (resp. $false$) on the environment associated to M.

2.2 Knowledge Layer

In our architecture, a knowledge base $\mathcal{B}$ is defined by a vocabulary, one or several bases of facts and a base of queries. We define these notions in terms of the structure layer defined above.

Definition 9. *A* vocabulary *is a tuple $V = (U = \bigcup_{1 \leq i \leq k} V_i)_{\leq_1, \dots \leq_q}$ where V_i are k sets of elements and $\leq_i$ are q preorders on U.*

Definition 10. *A* fact *is an ERGraph. A* base of facts *is a set of facts.*

Every ERGraph G respects $l_G : E_G \cup R_G \to L$ where L is constructed from the set U of elements of the vocabulary V of the knowledge base.

Definition 11. *A* query *is a couple* $Q = (q, \mathcal{C})$ *of a* λ*-ERGraph* q *and a Constraint system* $\mathcal{C}$*. A* base of queries *is a set of Queries.*

The λ-expression identifies the variables for which the values are searched when mapping the query with a fact. The answers to a query depend on the kind of EMapping used to query the base.

Definition 12. *Let* $Q = (((e_1, \ldots e_k), G), \mathcal{C})$ *be a query and* F *be a fact.* $A = (a_1, \ldots a_k)$ *is a* T-answer *to* Q *in* F *iff there exists an EMapping* M *of type* T *from* G *to* F *satisfying* $\mathcal{C}$ *such that* $M(e_i) = a_i$ *.*

The proof of a T-answer is the proof of the EMapping of that T-answer.

2.3 Language Layer

When considering a particular knowledge representation language its primitives are defined in terms of the knowledge layer of our model [1]. For instance, when considering the RDF/S language, an RDF triple $< s, p, v >$ is defined as a relation r_p in a ERGraph G such that $n_G(r_p) = (e_s, e_p, e_v)$. In the next section we propose an abstract machine based on this knowledge level of our model.

3 A Generic Query Language and its Interpreter

3.1 KGRAM's Generic Query Language

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

```
QUERY  ::= query(NODE *, EXP)
EXP    ::= QUERY
         | NODE | EDGE | PATH | FILTER
         | 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)
```

Overview of the Language Constructs. The QUERY expression in KGRAM's query language corresponds to the query defined in the knowledge layer of our graph-based KR model. Its parameters EXP and NODE represent the expression to evaluate and the nodes for which bindings are searched; they correspond in our KR model to the λ-ERGraph which composes a query. The constraint of the query corresponds to a FILTER expression embedded in the EXP expression.

Here is an example of an expression asking to query for authors and titles of documents. Its parameter EXP is an `and()` expression and its parameters NODE are `?x` and `?title`.

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

A QUERY expression also allows nested queries which 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 identifier of a node or an edge in a graph.

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

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

NODE, EDGE and FILTER are primitives that correspond to interfaces of KGRAM (c.f. section 4).

A PATH expression is a generalization of an EDGE expression. It allows us to query for paths of binary relations between two nodes in a graph. The `RegExp` parameter is a regular expression describing a set of relation paths. A path can consist of one relation or one relation path (`*` operator) any times or a sequence of relations or relation paths (`/` operator), with possible alternatives (`|` operator):

```
RegExp ::= label | RegExp '*' | RegExp'/'RegExp | RegExp'|'RegExp
```

Here is a query with a PATH expression to retrieve the elements of a list:

```
query({node('?y')}, path(rdf:rest*/rdf:first, node('?x'), node('?y')))
```

An `and` (resp. `union`) is available 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 restrains the search to the first solution retrieved. A `graph` expression can be used to specify the graph upon which the query is evaluated, otherwise a default graph is considered.

From Graph Homomorphism to SPARQL 1.1. Depending on the primitives we consider, we can define various query languages with KGRAM's abstract language. For instance the NODE and EDGE expressions enable us to express the query language of the Simple Conceptual Graph model [8, 3]. By including FILTER expressions, we can express conceptual graphs with constraints [2].

The expressions NODE, EDGE, FILTER, *and()*, *union()*, *option()* and *graph()* enable us to express the core of the SPARQL SELECT-WHERE query pattern

extended to n-ary relations. The *exist()* expression corresponds to the ASK query pattern of SPARQL. The *query()* and *path()* expressions capture the notions of nested query and relation path that can be found in SPARQL 1.1. SPARQL 1.1. agregates can also be expressed in KGRAM's language through pairs of a node and a filter as arguments of a *query()* expression.

3.2 KGRAM's Generic Interpreter

Natural Semantics of KGRAM's query language. Natural Semantics [6] is a semantics specification formalism originally used for programming languages where axioms and inference rules characterize each language construct. An inference rule is applied within an environment and produces one or several new environments. In the case of KGRAM, an environment represents a set of bindings of query variables with values. It corresponds to an ERProof in our KR model. The rules we established for KGRAM's query language describe the evolution of the environment (initially empty) during the evaluation of an expression building up a query. More precisely an expression in a query is evaluated in an environment comprising a valuation of the variables occuring in the query and bound to nodes of the graph which is queried. These bindings come from former evaluations of other expressions of the KGRAM language occuring in the query. The evaluation of an expression may produce several environments in case of multiple solutions. In that case the next expressions in the query are then evaluated in each of these environments. Once all the expressions of a query have been evaluated, each resulting environment corresponds to a retrieved solution.

Rules 1 and 2 describe the evolution of the environment during the evaluation of an expression of KGRAM's language. A list of environments is described by its first element *ENV* and the list of its other environments *LENV*. The evaluation of an expression *EXP* in each environment of the list *ENV.LENV* produces a list of environments *LENV'.LENV"* where *LENV'* comes from the evaluation of *EXP* in *ENV* and where *LENV"* comes from the evaluation of *EXP* in each environment of *LENV*, recursively applying rules 1 and 2.

$$\frac{ENV \vdash EXP \rightarrow nil \ \wedge \ LENV \vdash EXP \rightarrow LENV'}{ENV.\ LENV \vdash EXP \rightarrow LENV'} \tag{1}$$

$$\frac{ENV \vdash EXP \rightarrow LENV' \ \wedge \ LENV \vdash EXP \rightarrow LENV''}{ENV.\ LENV \vdash EXP \rightarrow LENV'.\ LENV''} \tag{2}$$

Rules 3 and 4 govern the evaluation of expressions for searching a NODE or an EDGE in a graph. They specify that the evaluation of such an expression in an environment *ENV* requires to compute the list of environments *LENV* capturing the possible matching of NODE or 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 the next subsection).

$$\frac{match(ENV \vdash NODE \rightarrow LENV) \ \wedge \ merge(ENV, LENV \rightarrow LENV')}{ENV \vdash NODE \rightarrow LENV'} \tag{3}$$

$$\frac{match(ENV \vdash EDGE \rightarrow LENV) \ \wedge \ merge(ENV, LENV \rightarrow LENV')}{ENV \vdash EDGE \rightarrow LENV'} \quad (4)$$

Rules 5 and 6 define the way to evaluate a FILTER expression. The rule base *eval* is relative to the evaluation of the boolean expression by which a FILTER expression is parameterized; it exploits the bindings of the query variables embedded in the current environment *ENV*. Rule 5 specifies that if this boolean expression is evaluated to *false* then an empty environment list (nil) is produced: there is no solution. Rule 6 specifies that otherwise the produced list contains a single element which is the current environment using the *list* operator.

$$\frac{eval(ENV \vdash F : false)}{ENV \vdash filter(F) \rightarrow nil} \quad (5) \qquad \frac{eval(ENV \vdash F : true)}{ENV \vdash filter(F) \rightarrow list\ ENV} \quad (6)$$

We will see in the next subsection that the rules associated to these three expressions of the query language — NODE, EDGE and FILTER — are the keystone of the algorithm of KGRAM interpreter.

Rules 7 and 8 define the way to evaluate a *and* expression. Rule 7 specifies that the evaluation of one argument expression of an expression *and* in the current environment produces a list of environments in which the other argument expression is evaluated. The list of environments produced by this last evaluation provides the result of the evaluation of the expression *and* as a whole. Rule 8 specifies that when the evaluation of the first argument expression produces an empty environment (no solution to this expression), it is useless to evaluate the other one (no solution to the expression *and* as a whole).

$$\frac{ENV \vdash A \rightarrow LENV \ \wedge LENV \vdash B \rightarrow LENV'}{ENV \vdash and(A, B) \rightarrow LENV'} \quad (7)$$

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash and(A, B) \rightarrow nil} \quad (8)$$

Rule 9 defines the way to evaluate a *union* expression. It specifies that the environments produced by the evaluation of a UNION expression in an environment *ENV* is the concatenation of those produced in *ENV* by evaluating each of the argument expressions of the *union* expression.

$$\frac{ENV \vdash A \rightarrow LENV \ \wedge \ ENV \vdash B \rightarrow LENV'}{ENV \vdash union(A, B) \rightarrow LENV \ . \ LENV'} \quad (9)$$

Rules 10 and 11 define the way to evaluate an *option* expression.

$$\frac{ENV \vdash A \rightarrow LENV}{ENV \vdash option(A) \rightarrow LENV} \quad (10) \qquad \frac{ENV \vdash A \rightarrow nil}{ENV \vdash option(A) \rightarrow list\ ENV} \quad (11)$$

Rules 12 and 13 define the way to evaluate a *not* expression.

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash not(A) \rightarrow list\ ENV} \quad (12) \qquad \frac{ENV \vdash A \rightarrow LENV}{ENV \vdash not(A) \rightarrow nil} \quad (13)$$

Rules 14 and 15 define the way to evaluate an *exist* expression.

$$\frac{ENV \vdash A \rightarrow nil}{ENV \vdash exist(A) \rightarrow nil} \quad (14) \qquad \frac{ENV \vdash A \rightarrow LENV}{ENV \vdash exist(A) \rightarrow list\ ENV} \quad (15)$$

Rule 16 defines the way to evaluate a *query* expression. It specifies that to evaluate *query(LNODE, EXP)*, bindings of the nodes in the argument *LNODE* of the expression are extracted from the current environment *ENV*: the *select* rule base specifies this operation. The expression *EXP* is then evaluated in the produced environment *ENV'*. The bindings of *LNODE* are then extracted from the environments *LENV'* produced by this evaluation (by using the same basic *select* rule base) and they are merged with the initial environment *ENV* (using the rule base *merge* already encountered in rules 3 and 4). A *query* expression "nested" into another one thus shares with the latter the only nodes contained in its list of node parameters.

$$\frac{\begin{array}{c} select(ENV \vdash LNODE \rightarrow ENV')\ \wedge ENV' \vdash EXP \rightarrow LENV' \wedge \\ select(LENV' \vdash LNODE \rightarrow LENV'')\ \wedge merge(ENV, LENV'' \rightarrow LENV) \end{array}}{ENV \vdash query(LNODE, EXP) \rightarrow LENV} \quad (16)$$

Rule 17 defines the way to evaluate a *graph* expression. It specifies that the result of the evaluation of *graph(G', EXP)* is the one of the evaluation of the expression *EXP* on *G'*. The argument *G'* is a node that represents the name of the current graph on which the expression is evaluated, i.e. where the answers to the query must be searched for. It is taken into account by the rule base *match* (in rules 3 and 4) that selects the graph for searching for matchings. This node may be a constant (a URI) or a variable for which the rule base *match* thus determines the bindings.

$$\frac{ENV, G' \vdash EXP \rightarrow LENV}{ENV, G \vdash graph(G', EXP) \rightarrow LENV} \quad (17)$$

The introduction of the expression *graph* in the language requires, like in rule 17, to add a node argument designating a graph to all the rules of the language (we deliberately do not mention this argument in our present presentation of the rules for the sake of simplicity).

The following five rules define the way to evaluate a *path* expression. The first rule specifies the evaluation of a *path* expression whose parameter is the elementary relation path pattern. The second rule specifies the evaluation of an expression $path(EXP_1/EXP_2, N_1, N_2)$. The third rule specifies the evaluation of an expression $path(EXP_1|EXP_2, N_1, N_2)$. The forth and fifth rules specify the evaluation of an expression $path(EXP^*, N_1, N_2)$.

$$\frac{ENV \vdash edge(P, N_1, N_2) \rightarrow LENV}{ENV \vdash path(P, N_1, N_2) \rightarrow LENV} \quad (18)$$

$$\frac{ENV \vdash path(EXP_1, N_1, N_i) \rightarrow LENV \wedge LENV \vdash path(EXP_2, N_i, N_2) \rightarrow LENV'}{ENV \vdash path(EXP_1\ /\ EXP_2,\ N_1,\ N_2) \rightarrow LENV'} \quad (19)$$

$$\frac{ENV \vdash path(EXP_1, N_1, N_2) \rightarrow LENV \wedge ENV \vdash path(EXP_2, N_1, N_2) \rightarrow LENV'}{ENV \vdash path(EXP_1 \mid EXP_2, N_1, N_2) \rightarrow LENV.LENV'} \quad (20)$$

$$\frac{ENV \vdash path(EXP, N_1, N_i) \rightarrow LENV \ \wedge LENV \vdash path(EXP^*, N_i, N_2) \rightarrow LENV'}{ENV \vdash path(EXP^*, N_1, N_2) \rightarrow LENV'} \quad (21)$$

$$\frac{ENV \vdash N_1 \rightarrow LENV \wedge bind(LENV \vdash N_2, N_1 \rightarrow LENV')}{ENV \vdash path(EXP^*, N_1, N_2) \rightarrow LENV'} \quad (22)$$

KGRAM's Evaluation Function. The core of KGRAM is its evaluation function which interprets KGRAM's abstract query language. Its algorithm implements the rules of Natural Semantics specifying the semantics of the query language and in particular those associated to the expressions NODE and EDGE. The operationalisation of these rules corresponds to the search of homomorphisms on labelled graphs: the environments produced by these rules represent the (partial) homomorphisms found between the expression of the query language and the data graph. The operationalization of the rules associated to expression FILTER corresponds to the search of homomorphisms with constraints.

KGRAM's algorithm is given on the next page. 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` responsible for the production of candidate nodes and edges of the data graph matching those of the query graph, (2) a `matcher` responsible for the matching of query and target nodes or edges, (3) an `evaluator` responsible for the evaluation of constraints (filters), (4) an environment manager `env` responsible for the storage in a stack structure of 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 expression). We will see in section 4 that the `producer`, the `matcher` and the `evaluator` called in this algorithm implement KGRAM's API. This ensures the independance of the interpreter of the query language from the data models and therefore the interoperability of KGRAM.

In the `switch` control instruction, the blocks labelled by NODE and EDGE implement the rules associated to the expressions NODE and EDGE of the query language and hence complete the current environment with node and edge bindings between the query and target graphs. The `getEdges` 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 then matched again with the query edge by the `matcher`. This enables to tune the semantics of the matching at KGRAM's level and therefore to handle possibly primitive producers which would exhibit inconvenient candidate edges or nodes. In the case where the

matching succeeds, the queried edges are added alternately 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()`.

```
eval(queryStack, i){
  if (queryStack.size() = i) {store(env); return;}
  exp = queryStack(i);
  switch(exp){
    case EDGE:
      for (Edge r : producer.getEdges(exp, env)){
         if (matcher.match(exp, r)){
           env.push(exp, r));
           eval(queryStack, i+1);
           env.pop(exp, r);}}
      break;
    case NODE:
      for (Node n : producer.getEdges(exp, env)){
         if (matcher.match(exp, n)){
           env.push(exp, n))
           eval(queryStack, i+1);
           env.pop(exp, n);}
      break;
    case FILTER:
      if (evaluator.test(exp, env)) eval(queryStack, i+1);
    ... }}
```

The double recursion suggested by the rules of natural semantics associated to the expressions `node` and `edge` — on the expression to evaluate and the environments to build — translates to the recursivity of the `eval` function and the iterative recording in a list of the complete homomorphisms built in `env`.

The FILTER block in the `switch` control instruction implements the rules 5 and 6 specifying the expression FILTER. KGRAM delegates the evaluation of filters (constraints) to an abstract filter evaluator `evaluator`. The `test` function of the latter takes as argument a filter to be evaluated and the current environment which acts as a variable binding environment. If the filter evaluates to true, the search for 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`, to enumerate new candidates and then evaluate the filter in other environments where it may succeed.

All the other rules of natural semantics specified for KGRAM's language have been implemented in the interpreter by specific blocks integrated to the backbone of the algorithm shown above: each expression has its own block. We do not detail them in this paper. Note that these are just some of the blocks of instructions that modify the stack of expressions `queryStack` to be evaluated.

When the partial homomorphism represented by the current environment cannot be completed, a recursive call of the `eval` function enables to backtrack

in the call stack and therefore to restore previous states of the stack `queryStack` and of the corresponding environment stack. The enumeration of candidate nodes or edges then continues in order to find new bindings in this state. In order to optimize the algorithm, we have refined backtrack (i.e. return to level n-1) with a mechanism that enables to backtrack directly at a lower level (e.g. n-2) in case of a local failure. This mechanism, called *backjump*, enables to return directly to an expression whose evaluation furnishes a new binding for at least one query node from the last expression that just failed. For this purpose, the `push` function of the `env` environment records, for each node binding, the position of the first expression in the expression stack that produces its first binding. The `backjump` fonction of the `env` environment is then able to compute the position in the expression stack where to backtrack when an expression fails: backjump occurs at the greatest bind index (highest in the stack) that may modify the binding of one of the nodes of current expression.

KGRAM's evaluation function is also optimized by using its ability to build homomorphisms alternatively by node search and edge search. This enables to optimize the evaluation of a query in the cases where some query nodes are known statically or some query edges have very few target candidates.

4 Querying and Mashup of Linked Data with KGRAM

The KGRAM interpreter comes with the application programming interface (API) it uses — and default implementations of it. In this section we describe the overall architecture of KGRAM API. Then we show how to build KGRAM-based applications to query and mashup linked data. A number of the applications we mention in this section have been captured in demos available on our site[3].

4.1 KGRAM Application Programming Interface

KGRAM interprets the expressions of its query language by using only abstract interfaces and hence remains independant of any graph implementation and any data structure.

Abstract Data Structures. The KGRAM interpreter accesses the queried data bases through an abstract API that hides the graph's structure and implementation. In other words, it operates on a graph abstraction by means of abstract structures and functions and it ignores the internal structure of the nodes and edges it manipulates to evaluate 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 of KGRAM's query language. As a result, KGRAM can process any kind of knowledge graph.

[3] http://www-sop.inria.fr/edelweiss/wiki/wakka.php?wiki=Demos

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 occuring in a given expression (and implementing the same APIs). It uses the KGRAM APIs of a node and edge matcher described bellow and thus ignores the way nodes or edges are matched.

A node and edge matcher implements the KGRAM *Matcher* interface. It is responsible for comparing node and edge labels. It implements the *match* semantic rule base occuring in the rules of natural semantics specifying the expressions NODE and EDGE of the query language. Depending on the *Matcher* implementation, the 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 specify the FILTER expression of the query language. 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. This *eval* function implements the *eval* rule base occuring in rules 5 and 6 of the operational semantics of KGRAM interpreter.

Genericity of the KGRAM Interpreter. The design of the KGRAM interpreter relies on interfaces. It is both independant of the concrete query language and data model. Access to data is mediated by an abstract producer and an abstract matcher. Moreover the evaluation of filters is delegated to an abstract filter evaluator. It is hence independant of the nature of the filters processed — which depends on the filter language implemented by the filter evaluator.

We have tested KGRAM's portability by implementing its interfaces *Node*, *Edge*, *Producer*, *Matcher* and *Evaluator* with both Corese[4] and Jena[5] [7].

KGRAM interfaces are designed in order to minimize the glue code. As a result, Corese's and Jena's portings to KGRAM have required quite a few source lines of code. Corese's porting was almost immediate because KGRAM was partly designed as an abstraction of the principles of Corese. Jena's porting has required less than 1000 source lines of code.

For the validation of KGRAM with one implementation or the other, we have used a RDF base comprising 25,000 triples and a base of 500 queries. In Corese's porting, KGRAM interprets its whole query language and queries RDF data implemented as conceptual graphs. In Jena's porting, KGRAM interprets the subset of its language corresponding to SPARQL 1.0 and queries RDF data.

We have also ported KGRAM on a new graph data structure which directly implements our ERGraph model. For the validation of this port of KGRAM, we have successfully implemented the W3C SPARQL 1.1 Query Test cases[6]. We are

[4] http://www-sop.inria.fr/edelweiss/software/corese/
[5] http://jena.sourceforge.net/
[6] http://www.w3.org/2009/sparql/docs/tests/

currently using this implementation in an application in genomics (*BioMarker*: RDF graphs in *Cytoscape* viewer) which manages 1.6 million triples and runs a workflow of 10 queries in 1 second on a laptop.

4.2 Querying and Mashup of Distributed Data

KGRAM relies on its `Producer` interface for the enumeration of data edges and nodes. This enables to seamlessly design a producer that enumerates edges coming from *several* graph stores. For this purpose, we have designed a metaproducer which implements the `Producer` interface and is an iterator of producers, each of which implements `Producer` and can enumerate edges and nodes from one graph store.

This metaproducer is used in the ISICIL project[7] to answer a usage scenario where RDF data is distributed over three servers for performance issues due to the size of the knowledge bases and their heterogeneity. Each RDF server is in charge of inferences on specific types of data: (1) social network and user profiles, online communities, activity tracking and trust model; (2) tag model, document metadata, terminologies, thesaurus; (3) web resource model with low level data such as MIME type, production context, format, duration, etc.

Some of the web applications developed in the ISICIL framework require to answer queries over data distributed on these three servers. We have configured a metaproducer iterating over three RDF producers, one for each server. They all are implementations of KGRAM's `Producer` interface based on Corese. The application we developped with KGRAM thus enables to mashup the data of these three RDF stores with SPARQL queries which evaluation produces an environment involving values from the three RDF stores.

Querying and Mashup of Heterogenous Data. The implementation of KGRAM's `Producer` interface by a metaproducer is also the key to mashup data with heterogenous knowledge models. It suffices that an implementation of the `Producer` interface is implemented for each knowledge model, with also different implementations of the `Node` and `Edge` interface, and that a metaproducer iterates over all of them. In that case, the matcher which is called by the interpreter for each candidate node or edge returned by the metaproducer is here to harmonize the semantics of the preliminary matchings of the producers.

By default, KGRAM is provided with an implementation of its `Evaluator` interface which handles XML Schema datatypes. Depending on the data to mashup, the evaluator must be able to compare values coming from different implementations of a common datatype or values from different datatypes — with a cast system.

Mashup of Knowledge Graphs with XML and Relational Data. In addition to the querying of heterogenous data distributed over several graph stores

[7] http://isicil.inria.fr/

by combining dedicated graph producers, KGRAM's language and interpreter also enable to mashup data coming from XML or relational database during the evaluation of a query expression over a data graph.

It is known that RDF can embed XML Literal values by means of the `rdf:XMLLiteral` datatype. Unfortunalely, SPARQL does not allow to query the content of this structured datatype. Moreover URIs of resources can denote XML documents the content of which may be interesting to query. For this purpose we introduced in KGRAM's language an extension of SPARQL to process XML data using XPath. It consists in (1) an `xpath` function that enables to apply an XPath expression to an XML Literal or to an XML document at a given URI and (2) an `unnest` function that enables to enumerate a collection of results as variable bindings in a subquery. These two functions occur in the abstract syntax of the KGRAM's query language as parameters of a FILTER expression. More precisely, in the filter language, they are two instances of `FunctionName` (see section 3.2). Let us consider the example below. The combination of functions `xpath` and `unnest` enables to query for book titles in an XML document designated by the `?doc` variable and to bind the retrieved values to the `?title` variable which is used in a query expression of KGRAM's language to retrieve both the authors and titles of documents.

```
select * where {
  ?doc c:author ?a
  {select unnest(xpath(?doc, '/book/title/text()')) as ?title where {}}
  ?doc c:title ?title}
```

Similarly, we also introduced in KGRAM's language a `sql` function that computes a SQL query on a relational database. Each row in the SQL query result is translated into a variable binding.

5 Conclusion

KGRAM is an abstract machine which interprets an abstract query language to query knowledge graph models. We presented here the formal semantics of its abstract graph langage and query language. Its query language is an abstraction of a generalization and extension of SPARQL which enables us to (1) handle most features of SPARQL 1.1. recommendation, (2) query not only RDF but also any knowledge graph model, (3) mashup XML or relational data. It comes with an application programming interface and a default implementation to develop semantic web applications for the querying and mashup of distributed and heterogenous data.

The limits of KGRAM lie in its capabilities to mashup heterogenous data. These are precisely relative to the value datatypes of the knowledge models. In the simplest case where the different models share the same datatype (like the XML schema datatype standard) the problem is reduced to handling the different implementations of the adopted datatypes by different producers connected to KGRAM's metaproducer. Otherwise a cast system between different

datatypes must be developed which can be a difficult problem. KGRAM comes with a default evaluator implementation that manipulates values as Java Object. This generic evaluator is provided with a proxy that is able to evaluate basic expressions by casting Object values to target values (e.g. integers, strings, etc.). This two-stage implementation is a first part of the solution to the problem of heterogenous datatypes: KGRAM's default evaluator interprets a default filter language without freezing the choice of the implementation of the values.

We are currently integrating optimizations in the interpreter's algorithm, such as those described in [4], e.g. heuristically sorting query edges or retrieving several connected edges at once instead of enumerating them each after the other. We are also considering the problem of query distribution. We intend to make KGRAM's metaproducer call the producers on which it iterates in separate threads to allow us to scale to large distributed bases on the web of linked data.

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*, volume 5113 of *Lecture Notes in Computer Science*, pages 297–310. Springer, 2008.
2. J.F. Baget and M.L. Mugnier. Extensions of Simple Conceptual Graphs: the Complexity of Rules and Constraints. *J. Artif. Intell. Res. (JAIR)*, 16:425–465, 2002.
3. M. Chein and M.L. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer London Ltd, 2009.
4. 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*, volume 4604 of *Lecture Notes in Computer Science*, pages 472–475. Springer, 2007.
5. O. Corby and C. Faron-Zucker. The KGRAM Abstract Machine for Knowledge Graph Querying. In *Proc. of IEEE/WIC/ACM International Conference on Web Intelligence, WI 2010*. IEEE Computer Society, 2010.
6. G. Kahn. Natural Semantics. In *Proc. of 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
7. B. McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.
8. J.F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.