RAL: an Algebra for Querying RDF

Flavius Frasincar Geert-Jan Houben Richard Vdovjak
Peter Barna
Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, the Netherlands
$\{$*flaviusf, houben, richardv, pbarna*$\}$*@win.tue.nl*

Abstract

To make the WWW machine-understandable there is a strong demand both for languages describing metadata and for languages querying metadata. The Resource Description Framework (RDF), a language proposed by W3C, can be used for describing metadata about (Web) resources. RDF Schema (RDFS) extends RDF by providing means for creating application specific vocabularies (ontologies). While the two above languages are widely acknowledged as a standard means for describing Web metadata, a standardized language for querying RDF metadata is still an open issue. Research groups coming from both industry and academia are presently involved in proposing several RDF query languages. Due to the lack of an RDF algebra such query languages use APIs to describe their semantics and optimization issues are mostly neglected. This paper proposes RAL, an RDF algebra suitable for defining (and comparing) the semantics of different RDF query languages and (at a future stage) for performing algebraic optimizations. After the definition of the data model we present the operators by means of which the model can be manipulated. The operators come in three flavors: extraction operators retrieve the needed resources from the input RDF model, loop operators support repetition, and construction operators build the resulting RDF model.

1. Introduction

The next-generation Web, called the Semantic Web (SW), aims at making its content not only machine-readable but also machine-understandable [5]. The final goal of the SW is to make machines fully exploit the Web knowledge in an uncentralized manner. The W3C standard called Resource Description Framework (RDF) [8, 24] is intended to serve as a metadata language for the Web and together with its extensions lays a foundation for the SW. It has a graph notation, which can be serialized in a triple notation (subject, predicate, object) or in an XML syntax.

Compared to XML, which is document-oriented, RDF takes into consideration a knowledge-oriented approach designed specifically for the Web. One of the advantages of RDF over XML is that an RDF graph depicts in a unique form the information to be conveyed while there are several XML documents to represent the same semantic graph. RDF's strength is based on its ability to define statements that assign values to resource properties.

While object-oriented systems are object-centered (properties are defined in a class context), RDF is property-centric, which makes it easy for anyone to “say anything about anything” [4], an architecture principle of the SW. In RDF, concepts from E-R modeling are being reused for web ontology modeling, a common understanding of topics that allows application interoperability [12].

RDF Schema (RDFS) [24] can be used to define application specific vocabularies. These vocabularies define taxonomies of resources and properties used by specific RDF descriptions. RDFS is designed as a flexible language to support distributed description models. Unlike XML DTD/Schema, RDFS does not impose a strict typing on descriptions (e.g. one can use new properties that were not present in the schema, a resource can be an instance of more than one class etc.). The set of primitive data types in RDF is left on purpose poorly defined as it is envisaged that RDFS will reuse the work done for data typing in XML Schema [6]. We do hope that future versions of RDFS will bring clarification regarding this subject and will eliminate several other shortcomings of the present specification (e.g. missing set collection, difficult literals handling etc.).

In the XML world there is already a winner in the quest for the most appropriate XML query language, i.e. XQuery [7]. As the SW initiative started not too long ago, its supporting technologies are still in their infancy. Research groups coming from both industry and academia are presently involved in proposing several RDF query languages. Due to the lack of an RDF algebra such query languages use APIs to describe their semantics and the optimization issues are mostly neglected. This paper proposes RAL, an RDF algebra suitable for defining (and comparing) the semantics of different RDF query languages and (at a future stage) for performing algebraic optimizations.

2. Related Work

At present there already exist a few RDF query languages but to our knowledge there is no full-fledged RDF algebra. The only algebraic description of RDF that we encountered so far is the RDF data model specification [28] done by Sergey Melnik (Stanford). It is based on triples and provides a formal definition of resources, literals, and statements. Despite being nicely defined, the specification does not include URIs, neglects the RDF graph structure, and does not provide operations for manipulating RDF models. Another formal approach, which aims not only at formalizing the RDF data model but also at associating a formal semantics to it, is the RDF Model Theory (RMT) [18]. It does not however, qualify as an algebraic approach but rather, as the name suggests, a model theoretic one. As RMT is currently being considered a main reference when it comes to RDF semantics, we tried to make our algebra (especially the data model part) compatible with RMT.

As implementation of RDF toolkits started before having an RDF query language, there are a lot of RDF APIs present today. Three main approaches for querying RDF (meta)data have been proposed.

The first approach (supported in the W3C working group by Stanford) is to view RDF data as a knowledge base of triples. Triple [33], the successor of SiLRI (Simple Logic-based RDF Interpreter) [11], maps RDF metadata to a Horn Logic (replacing Frame Logic) knowledge base. A similar approach is taken in Metalog [26], which matches triples to Datalog predicates (a subset of Horn logic). In this way one can query RDF descriptions at a high level of abstraction, i.e. at a logical layer that supports inference [17].

The second approach proposed by IBM builds upon the XML serialization of RDF. In the "RDF for XML" project (recently removed), an RDF API is proposed on top of the IBM AlphaWork's XML 4 Java parser. In the frame of the same project a declarative query language for RDF (RDF Query) [25] was created for which both input and output are resource containers. One of the nice features of this query language is that it proposes operators similar to the relational algebra, leaving the possibility to reuse some of the 25 years experience with relational databases. Unfortunately it fails to include the inference rules specific to RDF Schema, loosing description semantics.

Stefan Kokkelink goes even further with the second approach proposing RDF query and transformation languages that extend existing XML technologies. With this respect he defines RDFPath [22], similarly to XPath, for locating information in an RDF graph. The location step and the filter constructs were present also in XPath, but the primary selection construct is new. With the RDF graph being a forest, one needs to specify from which trees the selection will be made. RDFT is an RDF declarative transformation language a la XSLT [21], while RQuery, an RDF query language, is obtained by replacing XPath [3] with RDFPath in XQuery [7]. However this approach is not using the features specific for RDF, as the RDF Schema is being completely neglected.

The third approach (coming from ICS-FORTH in Greece) uses the RDF Graph Model for defining the RDF query language RQL [20]. It extends previous work on semistructured query languages (e.g. path expressions, filtering capabilities etc.) [14] with RDF peculiarities. Its strength lies in the ability to uniformly query both RDF descriptions and schemas. Compared to the previous approach it exploits the inference given in the RDF Schema (e.g. multiple classification of resources, taxonomies of classes/properties etc.) being the most advanced RDF query language proposed so far.

Other query languages have been proposed during the last years: Algae [31] (W3C) and rdfDB query language [16] (Netscape) as graph matching query languages. RDF query languages similar to rdfDB are: RDFQL [19], David Allsop's RDF query language [1], SquishQL [30], and RDQL [32] (HP Labs) an implementation of SquishQL on top of the Jena RDF API [27] of Brian McBride (HP Labs). Some other proposed RDF APIs are: Wilbur [23] (Nokia), the RDF API of Sergey Melnik [29] (Stanford), and Redland [2]. DQL, the query language for DAML+OIL Web ontology language [9] (built on top of RDF), is currently under development.

All the proposed approaches disregard the reconstruction of the output: they leave the output as a "flat" RDF container of input resources. An RDF algebra needs to take into account also the construction part, as the resulting RDF graph can contain new vertices and edges not present in the original RDF graph. This construction part is not only necessary to express RDF queries, but also for their optimization. Query optimization can be achieved not only in the extraction part but also in the construction part when the actual output is going to be produced.

3. Data Model

An RDF model is similar to a directed labeled graph (DLG). However, it differs from a classical DLG definition since it allows for multiple edges between two nodes. It also differs from a multigraph because the different edges between two nodes are not allowed to share the same label. The graph is not necessarily connected and it can contain cycles.

The graph nodes represent resources or literals (strings). Resources can be further classified as URI references or blank nodes. Each blank node (also called an anonymous resource) is unique in the graph despite the fact that it has no label associated to it. Non-blank nodes are labeled with resource identifiers or string values. The graph edges represent properties and are labeled by property names. Edges between different pairs of nodes may share the same label and the same property can be applied repetitively on a certain resource. This RDF feature enables multiple classification of resources, multiple inheritance for classes, and multiple domains/ranges for properties. Both resources and properties are first class citizens in the proposed RDF data model.

We identify the following sets: R (set of resources), U (set of URI references), B (set of blank nodes), L (set of literals), and P (set of properties). At RDF level the following holds for the above sets: $R = U \cup B$, $rdf{:}Property \in U$, $P \subset R$, $rdf{:}type \in P$, and U, B, and L are pair-wise disjoint.

The property $rdf{:}type$ defines the type of a particular resource instance. At RDF level any resource can be the target of $rdf{:}type$. RDF supports multiple classification of resources because $rdf{:}type$, (as any other property) can be repeated on a particular resource.

Definition 1. An RDF model M is a finite set of triples (statements)

$$M \subset R \times U \times (R \cup L)$$

Definition 2. The set of properties of an RDF model M is

$$P = \{p|(s,p,o) \in M \vee (p, rdf{:}type, rdf{:}Property) \in M\}$$

Formally the data model (graph model) corresponding to an RDF model M is

$$G = (N, E, l_N, l_E), l_N : N \to R \cup L, l_E : E \to P$$

using the following construction mechanism: for each $(s,p,o) \in M$ add nodes $n_s, n_o \in N$ (different only if $s \neq o$) with $l_N(n_s) = s$, $l_N(n_o) = o$, and add $e_p \in E$ as a directed edge between n_s and n_o with $l_E(e_p) = p$. In case that s and/or $o \in B$ then $l_N(n_s)$ and/or $l_N(n_o)$ are not defined. The function $l_N(.)$ is an injective partial function, while $l_E(.)$ is a (possibly non-injective) total function.

We use quotes for strings that represent literal nodes to make a syntactical distinction between them and URI nodes. A URI can be expressed by qualified names (e.g. $s{:}Painting$) or in absolute form (e.g. $http{:}//example.com/schema\#Painting$). Blank nodes do not have a proper identifier which implies that they can be queried only through a property pointing to them. Compared to XML, which defines an order between subelements, in RDF the properties of a resource are unordered unless they represent items in a sequence container. Not having the burden of preserving element order eases the definition of algebra operators and their laws.

Each node has three basic properties as described in Table 1. The id of a node represents the label associated to it. The nodes from the subset of resources that represent the blank nodes do not have an id associated to them. There are two $type$s of nodes: $rdfs{:}Resource$ and $rdfs{:}Literal$. The $nodeID$ gives the unique internal identifier of each node in the graph. $nodeID$ has the same value as the id for the nodes that have a label, but in addition it gives a unique identifier to the blank nodes. The internal identifier $nodeID$ is not available for external use, i.e. it is not disclosed for querying.

Table 1. Basic properties for nodes

Basic property	Result for resources	Result for literals
id	$l_N(u), u \in U$	$l_N(s), s \in L$
$type$	$Resource$	$Literal$
$nodeID$	internal ID	internal ID

Each edge has three basic properties as described in Table 2. Compared with nodes, which have unique identifiers, edges have a $name$ (label), which may be not unique. There can be several edges sharing the same $name$ but connecting different pairs of vertices. The $name$ of an edge is (lexically) identified with the id of the resource corresponding to that property. The $subject$ of an edge gives the resource node from which the edge is starting and $object$ returns the resource or literal node (i.e. the value of the property) where the edge ends.

Table 2. Basic properties for edges

Basic property	Result
$name$	$l_E(p), p \in P$
$subject$	$r, r \in R$
$object$	$o, o \in R \cup L$

Definition 3. Two non-blank nodes are considered to be equal if they have the same id. Two blank nodes are considered to be equal if they have the same (RDF) properties and the corresponding (RDF) property values are equal.

All equal non-blank nodes are internally mapped into one node in the graph.

Definition 4. Two graphs are considered to be equal if they differ only by re-naming the $nodeID$ of their blank nodes.

Note that two graphs that have all their nodes equal (node equality) may be not equal (graph inequality) if some corresponding non-blank nodes have different properties and/or different property values.

RDF Schema (RDFS) provides a richer modeling language on top of RDF. RDFS adds new modeling primitives by introducing RDF resources with an additional semantics. If one chooses to discard this special semantics, RDFS models can be viewed as (plain) RDF models. Figure 1 depicts graphically the RDF/RDFS primitives.

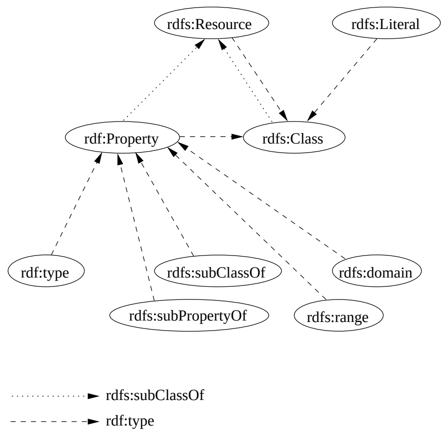

Figure 1. RDF/RDFS primitives

RDFS supports taxonomies of resources/properties using the inheritance mechanism at both class level (using the property $rdfs{:}subClassOf$) and property level (using the property $rdfs{:}subPropertyOf$). It also defines constraints (e.g. names to be used for properties, domain and range for properties etc.) that need to be fulfilled by RDF descriptions (later on called instances) in order to validate them according to the associated schema. RDFS provides a type system built on the following primitives: $rdfs{:}Resource$, $rdf{:}Property$, $rdfs{:}Class$, $rdfs{:}Literal$, $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $rdfs{:}domain$, and $rdfs{:}range$ The distinction between rdf and $rdfs$ namespaces to be used for different resources is more due to historical reasons (RDF was developed before RDFS) than due to semantical ones.

Every resource that has the $rdf{:}type$ equal to $rdfs{:}Class$, represents a type (or class) in the RDF(S) type system. Types can be classified as primitive types: $rdfs{:}Resource$, $rdf{:}Property$, $rdfs{:}Class$, $rdfs{:}Literal$ or user-defined types (resources defined explicitly by a particular RDF model to have the $rdf{:}type$ equal to $rdfs{:}Class$). The type of the resource $rdfs{:}Class$ is defined reflexively to be $rdfs{:}Class$. $rdfs{:}Class$ contains all the types, which is not the same thing as saying that it includes all the values (instances) represented by these types.

We extend the data model with the set C (set of classes). At RDFS level the following holds: $C \subset R$, $rdfs{:}Resource \in C$, $rdf{:}Property \in C$, $rdfs{:}Class \in C$, and $rdfs{:}Literal \in C$.

Definition 5. The set of classes of an RDF model M is

$$C = \{c | (c, rdf{:}type, rdfs{:}Class) \in M\}$$

The most general types are $rdfs{:}Resource$ and $rdfs{:}Literal$ which represent all resources and literals respectively. According to the data model these types are disjunctive. Subclasses of $rdfs{:}Resource$ are $rdfs{:}Class$ and $rdfs{:}Property$, $rdfs{:}Class$ representing all types (already stated above) and $rdfs{:}Property$ containing all properties. The distinction between properties and resources is not a clear cut one as properties are resources with some additional (edge) semantics associated to them. A property can be used repetitively between nodes (somehow similar to repeating a particular type in the definition of its instances) which justifies the existence of an $extent$ function (defined later on) for properties (as well as for classes). Moreover property instances can have the $rdfs{:}subPropertyOf$ defined in the same way as one can use the $rdfs{:}subClassOf$ for classes.

From all RDFS properties the most important ones are: $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $rdfs{:}domain$, and $rdfs{:}range$ (all being instances of $rdf{:}Property$). $rdfs{:}subClassOf$ and $rdfs{:}subPropertyOf$ are used to define inheritance relationships between classes and properties respectively. According to the RDF Test Cases [15] $rdf{:}subClassOf$ and $rdf{:}subPropertyOf$ can produce cycles, a useful mechanism if we think about class or property equivalence. A resource of type $rdf{:}Property$ may define the $rdfs{:}domain$ and the $rdfs{:}range$ associated to that property (i.e. the type of the subject/object nodes of the property edge in an RDF instance). Inspired by ontology languages, like OWL [10], $rdfs{:}domain$ and $rdfs{:}range$ can be multiply defined for one particular property and will have conjunctive semantics.

There is one particular class called $rdfs{:}Literal$ that represents all strings. Note that the RDF Model Theory [18] provides a more complex definition of the literal as a triplet (bit, character string, language tag), where the bit is used to flag if the character string represents an XML fragment or not. In the data model we simplify the literal definition considering just the character string from the above triple. Note that literals are not resources, i.e. one cannot associate properties to them. On the other hand there are resources that have type $rdfs{:}Literal$ and thus can have properties attached to them. Nevertheless one cannot say which literal this resource denotes. RDF defines also container classes: $rdf{:}Seq$, $rdf{:}Bag$, and $rdf{:}Alt$ to model ordered sequences, sets with duplicates, and value alternatives. The properties $rdf{:}rdf\_1$, $rdf{:}rdf\_2$, $rdf{:}rdf\_3$ etc. refer to the container members.

Each node representing a class has three schema properties as shown in Table 3. Schema properties associated to nodes are short notations (like a macro) for expressions doing the same computation based only on basic properties. The *type* of a class node is $Class$. The set of superclasses (classes from which the current class node is inheriting properties) is given by $subClassOf$. RDFS allows multiple inheritance for classes because $rdfs{:}subClassOf$ (as any other property) can be repeated on a particular class. The $extent$ of a class node is the set of all instances of this class.

Table 3. Schema properties for class nodes

Schema property	Result
$type$	$Class$
$subClassOf$	$S, S \subset C$
$extent$	$R^{'}, R^{'} \subset R$

Each node representing a property has five schema properties as shown in Table 4. The *type* of a property node is $Property$. The set of superproperties (properties which the current one is specializing) is given by $subPropertyOf$. Note that the domain or range of a superproperty should be superclasses for the domain or range, respectively, of the current property. The $domain$ and $range$ return sets of classes that represent the domain and the range, respectively, of the property node. The $extent$ of a node is the set of resource pairs linked by the current property (which is a subset of the Cartesian product between the associated domain and range extents).

Table 4. Schema properties for property nodes

Schema property	Result
$type$	$Property$
$subPropertyOf$	$S, S \subset Property$
$domain$	$D, D \subset Class$
$range$	$R, R \subset Class$
$extent$	$E, E \subset domain \times range$

One should note that we assume in the data model that there can be several edges having the same $name$ but linking different pairs of resources. All these properties can be seen as "instances" (abusing the "instance" term, previously referring to resource instances of a particular class) of the property node with the id equal to their common name.

In the absence of a schema, all RDF properties are of type $rdf{:}Property$ with domain equal to R and range equal to $R \cup L$. In this way one can define the $extent$ of an RDF property even if the property is not explicitly defined in a schema. In a schemaless RDF graph all resources are assumed to be of type $rdfs{:}Resource$.

The RDF Model Theory [18] defines the RDF-closure and RDFS-closure of a certain model M by adding new triples to the model M according to some given inference rules. We call the original model M the extensional data and the newly generated triples we call intensional data. There are two inference rules for RDF-closure and nine inference rules for RDFS-closure. The inference rules for RDF-closure are adding the $rdf{:}type$ (pointing to $rdf{:}Property$) for all properties in the model. Examples of inference rules for RDFS-closure are: transitivity of $rdfs{:}subClassOf$, transitivity of $rdfs{:}subPropertyOf$, $rdf{:}type$ inference based on a $rdf{:}type$ edge followed by an $rdfs{:}subClassOf$ edge etc. One should note that the output of these inference rules may trigger other rules. Nevertheless the rules will terminate for any RDF input model M, as there is only a finite number of triples that can be formed with the vocabulary of M.

Definition 6. An RDF model M is complete if it contains both its RDF-closure and RDFS-closure.

In the proposed data model we neglect reification and built-in properties like $rdfs{:}seeAlso$, $rdfs{:}isDefinedBy$, $rdfs{:}comment$, and $rdfs{:}label$ without loosing generality.

4. RAL

The purpose of defining RAL is twofold: to support the formal specification of an RDF query language and to enable algebraic manipulations for query optimization. RAL is an algebra for RDF defined from a database perspective, some of its operators being inspired by their relational algebra counterparts. We used a similar approach in developing XAL [13], an algebra for XML query optimization.

During the presentation of RAL operators we will use the RDF data from the example in Figure 2 as input. It is assumed that all operators know about the complete RDF model as it was defined in Section 3. That means that they all have the complete knowledge (extensional and intensional data) present in the given model. Variants of the proposed operators can be defined using the suffix "ˆ" which will make the operators neglect the intensional data, i.e. data derived by applying RDF(S) inference rules to the input model is neglected (similar to the RQL 'strict interpretation'). In order to simplify Figure 2 we chose to present only the extensional data and just one intensional data element given by the inferred edge $rdf{:}type$ between $r4$ and $Creator$.

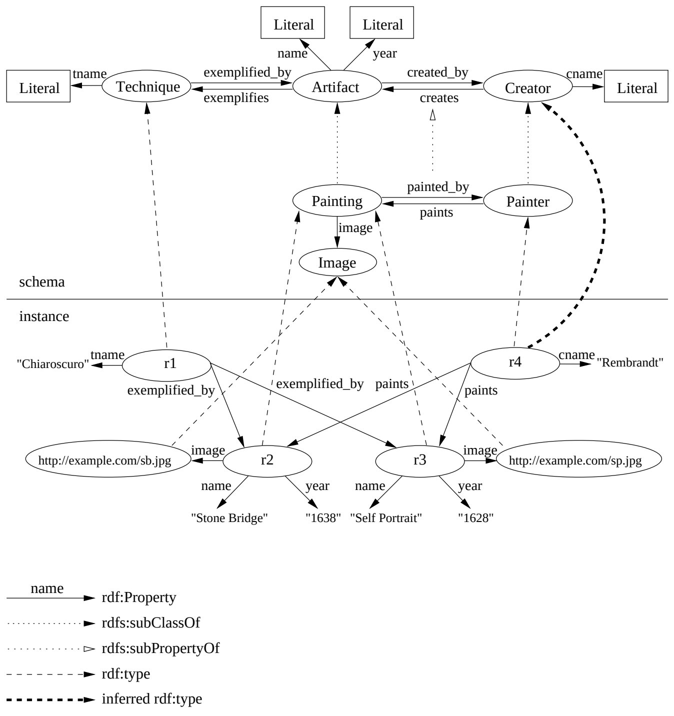

Figure 2. Example schema and instance

Figure 2 is an excerpt from the RDF schema and RDF instance of some Web data describing different painting techniques. For reasons of simplicity we consider only one painting technique ("$Chiaroscuro$"), one painter ("$Rembrandt$"), and two paintings of the same painter ("$StoneBridge$" and "$SelfPortrait$"). The schema does not present the RDFS primitives $rdfs{:}Resource$, $rdf{:}Property$, $rdfs{:}Class$, and $rdfs{:}Literal$ from which all the resources and literals are derived.

We define RDF collections to be sets of nodes (resources/literals). All RAL operators are closed for collections, which implies that RAL expressions can be easily composed. The operators come in three flavors: extraction operators retrieve the needed resources from the input RDF model, loop operators support repetition, and construction operators build the resulting RDF model.

The general form of the operators is

$$o[f](x_1, x_2, \ldots x_n : expression)$$

Informally, this means the following. For each binding of x to an element in the input collection, given by $x_1, x_2, \ldots x_n$, $f(x)$ is computed. f is a function that allows to refer to basic/derived properties or to one of the proposed operators. Based on the semantics of operator o a partial result for the application of o to $f(x)$ is computed. The operator result is obtained by combining (set union) all partial results. All unary operators use this implicit mechanism, the map operator, to compute the result. In the operator's general form, f is optional. One should note that an n-ary operator can be reduced to a unary one by having as input a sequence type.

RAL operators are defined to work on any RDF descriptions, with or without an explicit schema. Note that implicitly there is always a default schema based on the RDFS primitives $rdfs{:}Resource$, $rdf{:}Property$, $rdfs{:}Class$, and $rdfs{:}Literal$. These RDFS primitives can be used to retrieve a particular schema in case that such information is not known in advance. Once the application schema is known, one can formulate queries to return instances from the input RDF model.

4.1. Extraction Operators

The extraction operators retrieve the resources/literals of interest from the input collection of nodes. If the operator is not defined on nodes that represent literals, these nodes are simply neglected.

In the examples that illustrate the operators we will use expressions representing resources from the example RDF model m of Figure 2. The expression c represents the collection (set) of all resources present in model m.

Projection

$$\pi[re\_name](e : expression)$$

The input of the projection is a collection of nodes (specified by the expression e) and the projection operator computes the values (objects) of the properties with a name given by the regular expression re_name over strings. The string that matches all names is denoted by $\#$.

Example 1. $\pi[(P|p)aint[s]\#](r4)$ returns the collection of resources in our RDF graph painted by $r4$ (Rembrandt) (i.e. $r2$ and $r3$).

Example 2. $\pi[rdf{:}type](r4)$ returns the resource(s) representing the type of $r4$ (i.e. $Painter$).

Note that in these examples we talk about resources, where actually the output of the operators are collections of resources.

Selection

$$\sigma[condition](e : expression)$$

The $condition$ is a Boolean function that uses as constants URIs and/or strings. The operators allowed in the $condition$ are RAL operators, the usual comparison operators $(=, >=, <=, <, >, <>)$, and logical operators (and, or, not). The input of the selection is a collection of nodes and the operator selects only the nodes that fulfill the $condition$.

Example 3. $\sigma[\pi[tname] = \text{“}Chiaroscuro\text{”}](c)$ is a selection operation applied to the collection c of all resources from Figure 2. The expression returns the resource(s) representing the painting technique with the name “$Chiaroscuro$” (i.e. $r1$).

Example 4. The expression $\sigma[\pi[rdf{:}type] = Creator](c)$ returns resources with the value of $rdf{:}type$ being $Creator$ (i.e. $r4$, a resource of type $Painter$ with $Painter$ being a subclass of $Creator$).

Example 5. The expression $\hat{\sigma}[\pi[rdf{:}type] = Creator](c)$ (different from the selection in the previous example, as it asks for the use of only the extensional data) returns the empty collection, as the inferred $rdf{:}type$ of $r4$ (i.e. $Creator$) from Figure 2 will not be available to the operator.

Cartesian Product

$$(x : expression) \times (y : expression)$$

The Cartesian product takes as input two collections of nodes on which it performs the set-theoretical Cartesian product. Each pair of nodes builds an anonymous resource that has all the properties of the original resources. Thus, such a resource will have all the types of the original two resources (RDF multiple classification of resources). The final output is the collection of all the anonymous resources. Note how for binary operators we also use an infix notation.

Example 6. The expression $\sigma[\pi[rdf{:}type] = Technique](c) \times \sigma[\pi[rdf{:}type] = Painter](c)$ returns an anonymous resource having all the properties of $r1$ and $r4$. As a consequence this anonymous resource has both types $Technique$ and $Painter$.

Join

$$(x : expression) \bowtie [condition]\ (y : expression)$$

The join expression is defined to be a Cartesian product followed by a selection, so equivalent to

$$\sigma[condition](x \times y)$$

The expression has as input two collections of resources that have their elements paired only if they fulfill the $condition$ (relating the left and right operands). Anonymous resources are built for each such pair. The output is the collection of all the anonymous resources.

Example 7. $(x : \sigma[\pi[rdf{:}type] = Technique](c)) \bowtie [\pi[exemplified\_by](x) = \pi[paints](y)](y : \sigma[\pi[rdf{:}type] = Painter](c))$ returns an anonymous resource having all the properties of $r1$ and $r4$. In case that there would have been painters who didn't use the “$Chiaroscuro$” technique, these painters would not be paired with this technique in the resulting anonymous resources.

Union

$$(x : expression) \cup (y : expression)$$

The union operator combines two input collections of nodes reflecting the set-theoretical union.

Difference

$$(x : expression) - (y : expression)$$

The difference operator returns the nodes present in the first input collection but not in the second input collection.

Intersection

$$(x : expression) \cap (y : expression)$$

The intersection operator returns the nodes present in both input collections.

4.2. Loop Operators

Loop operators are used to control the repetitive application of a certain function or operator. They express repetition at input or function/operator level.

Map

$$map[f](e : expression)$$

The map operator is defined as

$$\cup(f(e_1), f(e_2), ...f(e_n))$$

if the collection e contains the elements $e_1, e_2, ...e_n$. So, the map operator expresses repetition at input level. The results of applying the function/operator f to each element in the input collection are combined (set union) to obtain the final result. All unary construction operators have an implicit map operator associated with them.

Example 8. $map[id](c)$ computes the labels of all the non-blank nodes in the (input) model, i.e. of all resources having an id property.

Kleene Star

$$*[f](e : expression)$$

The Kleene star operator is defined as

$$e \cup f(e) \cup ...f(f(...(f(f(e))))) \cup ...$$

So, the Kleene star operator expresses repetition at function/operator level. It repeats the application of the function/operator f on the given input possibly infinite number of times. For each iteration the result is obtained by combining (set union) the output of the function/operator application on the input with the input. If after an iteration the result is the same as the input, a fixed point is reached and the repetition stops. In order to ensure termination, a variant of this operator that specifies the number of iterations n is defined below:

$$*[f, n](e : expression)$$

Note that the map operator does not include the input in the result, while the Kleene star operator does.

Example 9. $map[id](*[\pi[rdfs{:}subClassOf]](Painting))$ gives the id of all ancestor classes in the type hierarchy starting with $Painting$. For our example the result will contain three resources representing the types $Artifact$, $Painting$, and $rdfs{:}Resource$. If there were loops made by the $rdfs{:}subClassOf$ property in the input model, the above example would still terminate. The fact that the input model has a finite number of classes implies that at a certain moment a fixed point is reached (we obtain the same output collection as for the previous iteration) and thus the Kleene star operator terminates.

4.3. Construction Operators

Querying an RDF model implies not only extracting interesting resources/literals from the input but also building new resources/literals and associating new properties between resources/properties.

Before inserting new nodes/edges, the RDF constraints are checked. If these constraints are not met, the operation aborts. Examples of RDF constraints are: the uniqueness of resource identifiers, the value of $rdf{:}type$ cannot be a literal, literals cannot have properties etc.

Create Node

$$node[type, id]()$$

The create node operator adds a new node to the graph. The input collection is not used in the operator semantics. The type of the node, specified by $type$, is a resource of type $Class$. The id is a resource identifier if the node represents a resource, or a string if the node represents a literal. As a side effect, an edge representing the $type$ property is added between the created resource and its associated type resource. All resources and literals need to define their type (even if it is one of the RDFS primitives $rdfs{:}Resource$ or $rdfs{:}Literal$). If the id operand is empty a blank node is constructed. For these blank nodes the system assigns unique $nodeID$s. The create node operator returns the created node (a collection containing one node).

Example 10. $node[Painter,]()$ creates a blank node of type Painter, while $node[Literal, \text{“}Caravagio\text{”}]()$ creates a Literal node representing the string “$Caravagio$”.

Create Edge

$$edge[name, subject](object : expression)$$

The create edge operator adds new edges (properties) to the graph. The name (label) of the edges, as specified by $name$, is the id of a resource of type $Property$ (property resource). The $subject$ and the $object$ must have types complying with the domain and the range of the associated (by name) property resource. The $subject$ is a node (or singleton collection) in the graph and the $object$ is a collection of nodes. The edges are between the $subject$ node and each of the nodes in the $object$ collection. The create edge operator returns the subject node (a collection containing one node).

Example 11. If $n1$ and $n2$ are the two nodes constructed in Example 10, $n1$ denoting the blank node and $n2$ denoting the literal node, $edge[name, n1](n2)$ creates an edge labeled $name$ between the nodes $n1$ and $n2$.

5. Conclusions

RAL is an RDF algebra defined to support the formal specification of an RDF query language. It presents a set of operations to be used in both the extraction and construction parts of a formally defined RDF query language. It is one of the first RDF algebras developed from a database perspective. Compared with existing RDF query languages, the construction phase is not neglected and is part of the language specification.

As future work we will analyze the expressive power of RAL with respect to existing RDF query languages and its completeness. Comparing the expressive power of RAL to that of other algebras, like relational algebra or object algebra, gives some insight into the real strength of the language, but the true test is the comparison with existing languages (implementations) for RDF: RQL is the prime candidate.

We would like also to investigate optimization laws that will enable algebraic manipulations for query optimization. The lack of order (between resources) in RDF models and RAL collections, as well as the simplicity and composability of RAL operators (similar to the relational algebra ones) seem to foster the definition of RAL optimization laws. A translator from a popular RDF query language (e.g. RQL) to RAL and a RAL engine will enable us to experiment with different aspects of RDF query optimization.

References

[1] D. Allsopp, P. Beautement, J. Carson, and M. Kirton. Towards semantic interoperability in agent-based coalition command systems. In *The Emerging Semantic Web - Selected Papers From The First Semantic Web Working Symposium*. IOS Press, 2002.

[2] D. Beckett. Redland rdf application framework. `http://www.redland.opensource.ac.uk`.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. W3C Working Draft 16 August 2002. `http://www.w3.org/TR/xpath20/`.

[4] T. Berner-Lee. What the semantic web can represent. W3C 1998. `http://www.w3.org/DesignIssues/RDFnot.html`.

[5] T. Berners-Lee. *Weaving the Web*. Harper San Francisco, 1999.

[6] P. V. Biron and A. Malhotra. Xml schema part 2: Datatypes. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-2/`.

[7] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. W3C Working Draft 16 August 2002. `http://www.w3.org/TR/xquery/`.

[8] D. Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3C Working Draft 30 April 2002. `http://www.w3.org/TR/rdf-schema/`.

[9] D. Connolly, F. van Harmelen, J. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Daml+oil (march 2001) reference description. W3C Note 18 December 2001. `http://www.w3.org/TR/daml+oil-reference`.

[10] M. Dean, D. Connoly, F. van Harmelen, J. Hendler, J. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language 1.0 reference. W3C Working Draft 29 July 2002. `http://www.w3.org/TR/2002/WD-owl-ref-20020729/`.

[11] S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for rdf. In *The W3C Query Languages Workshop*, 1998. `http://www.w3.org/TandS/QL/QL98/pp/queryservice.html`.

[12] S. Decker, S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and J. Harrocks. The semantic web: The roles of xml and rdf. *IEEE Internet Computing*, 4(5):63–74, 2000.

[13] F. Frasincar, G.-J. Houben, and C. Pau. Xal: an algebra for xml query optimization. In *Database Technologies 2002, Thirteenth Australasian Database Conference*, volume 5 of *Conferences in Research and Practice in Information Technology*, pages 49–56. Australian Computer Society Inc., 2002.

[14] C. R. G. G., B. Douglas, B. Mark, E. Jeff, J. David, R. Craig, S. Olaf, S. Torsten, and V. Fernando. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[15] J. Grant and D. Beckett. Rdf test cases. W3C Working Draft 29 April 2002. `http://www.w3.org/TR/rdf-testcases/`.

[16] R. V. Guha. Rdfdb query language. `http://web1.guha.com/rdfdb/query.html`.

[17] R. V. Guha, O. Lassila, E. Miller, and D. Brickley. Enabling inferencing. In *The W3C Query Languages Workshop*, 1998. `http://www.w3.org/TandS/QL/QL98/pp/enabling.html`.

[18] P. Hayes. Rdf model theory. W3C Working Draft 29 April 2002. `http://www.w3.org/TR/rdf-mt`.

[19] Intellidimension Inc. Rdfql query language reference. `http://www.intellidimension.com/RDFGateway/Docs/querying.asp`.

[20] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying rdf descriptions for community web portals. In *17iemes Journees Bases de Donnees Avancees*, pages 133–144, 2001.

[21] M. Kay. Xsl transformations (xslt) version 2.0. W3C Working Draft 16 August 2002. `http://www.w3.org/TR/xslt20/`.

[22] S. Kokkelink. Transforming rdf with rdfpath. Working Draft, 2001. `http://zoe.mathematik.Uni-Osnabrueck.DE/QAT/Transform/RDFTransform.pdf`.

[23] O. Lassila. Enabling semantic web programming by integrating rdf and common lisp. In *The First Semantic Web Working Symposium*, pages 403–410. Stanford, 2001.

[24] O. Lassila and R. R. Swick. Resource description framework (rdf) model and syntax specification. W3C Recommendation 22 February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

[25] A. Malhotra and N. Sundaresan. Rdf query specification. In *The W3C Query Languages Workshop*, 1998. `http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html`.

[26] M. Marchiori and J. Saarela. Query + metadata + logic = metalog, 1998. `http://www.w3.org/TandS/QL/QL98/pp/metalog.html`.

[27] B. McBride. Jena: Implementing the rdf model and syntax specification. In *Second International Workshop on the Semantic Web*, 2001. `http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-40/mc%bride.pdf`.

[28] S. Melnik. Algebraic specification for rdf models. Working Draft, 1999. `http://www-diglib.stanford.edu/diglib/ginf/WD/rdf-alg/rdf-alg.pdf`.

[29] S. Melnik. Rdf api draft, 2001. `http://www-db.stanford.edu/~melnik/rdf/api.html`.

[30] L. Miller. Inkling: Rdf query using squishql, 2002. `http://swordfish.rdfweb.org/rdfquery`.

[31] E. Prud'hommeaux. Algae howto. W3C, 2002. `http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html`.

[32] A. Seaborne. Rdql - a data oriented query language for rdf models. HP Labs. `http://www.hpl.hp.com/semweb/rdql.html`.

[33] M. Sintek and S. Decker. Triple - an rdf query, inference, and transformation language. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2002.