Preface

Preface must be added here.

Place, Month Year *First Name, Surname*

Table of Contents

1. RQL: A Functional Query Language for RDF
Gregory Karvounarakis, Aimilia Magkanaraki, Sophia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Karsten Tolle 1

1.1 Introduction 1
1.2 The RQL Data Model 2
1.2.1 Triple vs. Graph-based RDF/S Representation 4
1.2.2 A Formal Data Model for RDF 7
1.2.3 A Type System for RDF 10
1.3 The RDF Query Language: RQL 14
1.3.1 Basic Querying Functionality 14
1.3.2 RQL Filters 19
1.3.3 Schema Navigation 20
1.3.4 Data Navigation 24
1.3.5 Combining Schema and Data Navigation 25
1.4 Summary and Conclusions 29
References 29

Author Index 31

Subject Index 32

List of Contributors

S. Alexaki,
Institute of Computer Science
FORTH
Vassilika Vouton
P.O. Box 1385, GR 71110
Heraklion, Crete
Greece

V. Christophides
Institute of Computer Science
FORTH
Vassilika Vouton
P.O. Box 1385, GR 71110
Heraklion, Crete
Greece

G. Karvounarakis
Institute of Computer Science
FORTH
Vassilika Vouton
P.O. Box 1385, GR 71110
Heraklion, Crete
Greece

A. Magkanaraki
Institute of Computer Science
FORTH
Vassilika Vouton
P.O. Box 1385, GR 71110
Heraklion, Crete
Greece

D. Plexousakis
Institute of Computer Science
FORTH
Vassilika Vouton
P.O. Box 1385, GR 71110
Heraklion, Crete
Greece

M. Scholl
INRIA-Rocquencourt
78153
Le Chesnay Cedex
France

1. RQL: A Functional Query Language for RDF

Gregory Karvounarakis[1], Aimilia Magkanaraki[1], Sophia Alexaki[1], Vassilis Christophides[1], Dimitris Plexousakis[1], Michel Scholl[2], and Karsten Tolle[3]

[1] Institute of Computer Science, FORTH, Greece
email: {gregkar, aimilia, alexaki, christop, dp}@ics.forth.gr
[2] CEDRIC/CNAM, France
email: scholl@cnam.fr
[3] Johann Wolfgang Goethe-University
email: tolle@dbis.informatik.uni-frankfurt.de

Although real-scale Semantic Web applications, such as Knowledge Portals and E-Marketplaces, require the management of voluminous resource metadata, sufficiently expressive declarative languages for metadata created according to the W3C RDF/S standard[1] are still missing. Answering to this need, we have designed a typed, functional query language, called *RQL*, whose novelty lies in its ability to smoothly combine schema and data querying. The purpose of this chapter is to present *RQL*'s formal data model and type system and illustrate its expressiveness by means of exemplary queries. *RQL*'s formal foundations capture the RDF/S modeling primitives and provide a well-founded semantics for a declarative query language involving recursion and functional composition over complex description graphs.

1.1 Introduction

In the next evolution step of the Web, termed the *Semantic Web* [1.5], vast amounts of information resources (data, documents, programs, etc.) will be made available along with various kinds of descriptive information, i.e., *metadata*. The **Resource Description Framework** (RDF) [1.22, 1.8] constitutes part of the activity coordinated by the World Wide Web Consortium (`W3C`) for the management (encoding, exchange and process) of metadata as any other Web data. The objective of RDF is to enable the definition of resource descriptions in a formal, interoperable and humanly readable way via appropriate languages, without making any assumption about the application domain or the structure of the described information resources. More precisely, RDF provides i) a *Standard Representation Language* [1.22] for metadata based on *directed labeled graphs*, in which nodes are called *resources* (or *literals*) and edges are called *properties*; ii) a *Schema Definition Language* (RDFS) [1.8] for creating vocabularies of labels for these graph nodes (called *classes*) and

[1] For brevity, we denote as *RDF/S* the RDF Model&Syntax and Schema Specifications in their whole

edges (called *property types*); and iii) an *XML* [1.7] *syntax* for expressing metadata and schemas in a form that is both humanly readable and machine understandable. The most distinctive feature of RDF is its ability to *superimpose* several descriptions for the same information resources in a variety of application contexts (e.g., advertisement, recommendation, copyrights, content rating, push channels, etc.). Furthermore, metadata definitions can be exchanged, reused and extended, thus facilitating the automatic processing of resource descriptions on the Web.

Yet, the representation alone of resource metadata is not enough. Human information consumers as well as Semantic Web-aware agents have to use and query them. It becomes evident that managing voluminous *RDF description bases* and *schemas* with existing low-level APIs[2] (mostly file-based) does not ensure fast deployment and easy maintenance of Semantic Web applications [1.3]. Still, we want to benefit from database technology in order to support *declarative access* and *logical/physical RDF data independence*. In this way, Semantic Web applications have to specify in a high-level language only *which* resources need to be accessed, leaving the task of determining *how* to efficiently store or access the descriptions to the underlying *RDF database engine*.

The design of *RQL* [1.21, 1.20] has been motivated by the above issues. Based on the formal graph data model presented in Section 1.2.2 and on the type system presented in Section 1.2.3, *RQL* defines a set of basic queries and iterators, which can be used to build new complex queries, as demonstrated in Section 1.3. The smooth combination of *RQL* schema and data path expressions is a key feature for satisfying the needs of a wide-spectrum of Semantic Web applications.

1.2 The RQL Data Model

The RDF/S data model is based on the notion of "resource". Everything, concept or object, available on the Web or not, can be modelled as a resource uniquely identified by a *Universal Resource Identifier* (URI) [1.4]. In general, we can distinguish resources into *tokens*, *classes* and *metaclasses* by taking into account the abstraction layer to which they belong, i.e., their ability to contain members. More specifically, we can distinguish three abstraction layers. In the lower abstraction layer, the *data layer*, we have the descriptions of individual information resources. These descriptions can be stated in XML [1.7]. For instance, lines 24–28 of the example below illustrate how one can state in RDF that (a) the resource with URI `&r1` is both (i.e., *multiple classification*) a *Cubist* (lines 24–27) and *Sculptor* (line 28) and (b) it has

[2] For example, **GINF** (www-db.stanford.edu/~melnik/rdf/api.html), **RADIX** (www.mailbase.ac.uk/lists/rdf-dev/1999-06/0002.html) and **Jena** (www-uk.hpl.hp.com/people/bwm/RDF/jena)

two properties (i.e., *attribution*) *paints* with values the resources `&r2` and `&r3`, which are instances of *Painting* (lines 25 and 26). It should be stressed that the same RDF resource description can be expressed in a variety of equivalent syntactic forms using, for instance, XML attributes (like *paints*) instead of XML elements, flat (as *Sculptor*) versus nested XML elements (as *Cubist*), etc. This fact motivates further the need for an RDF/S query language abstracting the various XML syntactic variations of resource descriptions.

```
1 <rdfs:Class rdf:ID="Artist"/>
2 <rdfs:Class rdf:ID="Artifact"/>
3 <rdf:Property rdf:ID="creates">
4 <rdfs:domain rdf:resource="#Artist"/>
5 <rdfs:range rdf:resource="#Artifact"/>
6 </rdf:Property>
7 <rdfs:Class rdf:ID="Painter"/>
8 <rdfs:subClassOf rdf:resource="#Artist"/></rdfs:Class>
9 <rdfs:Class rdf:ID="Painting">
10 <rdfs:subClassOf rdf:resource="#Artifact"/></rdfs:Class>
11 <rdf:Property rdf:ID="paints">
12 <rdfs:subPropertyOf rdf:resource="#creates"/>
13 <rdfs:domain rdf:resource="#Painter"/>
14 <rdfs:range rdf:resource="#Painting"/>
15 </rdf:Property>
16 <rdfs:Class rdf:ID="Sculptor">
17 <rdfs:subClassOf rdf:resource="#Artist"/></rdfs:Class>
18 <rdfs:Class rdf:ID="Sculpture">
19 <rdfs:subClassOf rdf:resource="#Artifact"/></rdfs:Class>
20 <rdf:Property rdf:ID="technique">
21 <rdfs:domain rdf:resource="#Painting"/>
22 <rdfs:range rdf:resource=
          "http://www.w3.org/2001/XMLSchema#string"/>
23 </rdf:Property>
24 <Cubist rdf:about="&r1">
25 <paints><Painting rdf:about="&r2"/></paints>
26 <paints><Painting rdf:about="&r3"/></paints>
27 </Cubist>
28 <Sculptor rdf:about="&r1"/>
```

To accommodate the definition of valid description labels, RDF is enhanced with a Schema Definition Language (RDFS) [1.8] at a higher abstraction layer which can also be expressed in XML. At the *schema layer*, *classes* represent abstract entities referring collectively to sets of similar tokens and *properties* represent attributes or relationships among classes. In lines 3–6 of the above example, we can see the declaration of the property (*rdf:Property*) *creates*, having the classes (*rdfs:Class*) *Artist* (line 1) and *Artifact* (line 2) as its domain (*rdfs:domain*) and range (*rdfs:range*) respectively. Furthermore, using the the core RDF/S property *rdfs:subClassOf* (or *rdfs:subPropertyOf*), labels can be organized into taxonomies carrying inclusion semantics. Note that, in contrast to object-oriented data models [1.11], properties can also be organized into subsumption hierarchies. For example, class *Painter* is a subclass of class *Artist* (lines 7–8), while property *paints* refines the prop-

erty *creates* (lines 11–15). Although not illustrated in our example, *multiple specialization* is also possible.

In a nutshell, properties can be *inherited* (e.g., the property *creates* can be used for the description of resource `&r1`), are *unordered* and *optional* (e.g., the property *technique* is not used) and they can be *multi-valued* (e.g., we have two *paints* properties). Moreover, resources can be *multiply classified* under different schema constructs using the basic RDF property *rdf:type* (e.g., `&r1`). Due to this mechanism, we can describe a resource using properties from several classes not necessarily related through a subsumption relationship. This way, a resource can be described from many "viewpoints".

The upper abstraction layer, the *RDF/S layer* or *metaschema layer* is used to group schema entities into semantic units called *metaclasses*. We can distinguish between the *metaclasses of classes* (e.g., *rdfs:Class*) and *metaclasses of properties* ((*rdf:Property*). As we will see later in Figure 1.1, in order to introduce *user-defined* metaclasses, the core RDF/S metaclasses can also be refined using the RDF/S property *rdfs:subClassOf*.

Finally, the uniqueness of (meta)schema labels and the ability to reuse labels from several schemas is ensured by the XML namespace facility [1.6][3]. With the use of an XML namespace, descriptive terms (i.e., (meta)class or property names) are uniquely identified as normal web resources by a URI composed from their names prefixed with the namespace of their schema (e.g., *ns1#Artifact*).

1.2.1 Triple vs. Graph-based RDF/S Representation

Choosing a good data model to represent RDF/S descriptions and schemas is a core choice in order to design and formally define an RDF/S manipulation language. In particular, a query language describes in a declarative fashion, the mapping from an input instance of the data model to an output instance of the data model. In the previous section, we have seen that a direct usage of the XML syntax (and its underlying tree data model) in order to represent RDF/S (meta)data is not appropriate, since semantically equivalent descriptions may have several XML serializations. In this section, we compare two candidate RDF/S representations based on *triple* and *graph* models.

The *triple*-based model, provides a flat relational representation of RDF/S schemas and resource descriptions. More precisely, each RDF/S graph (i.e., a *data* or *schema description*) is represented as a set of edges, called *statements*. A *statement* is composed of a named edge (a property) and two end nodes (a resource and a value). Each statement can be represented by a *triple* having a *subject* (e.g., `&r1`), a *predicate* (e.g., `fname`), and an *object* (e.g., "Pablo"). The set of all statements referring to the same subject

[3] The prefixes *rdf* and *rdfs* are used to denote the namespaces where the basic RDF and RDFS modeling constructs (either at the schema or metaschema layers) are respectively defined

(i.e., the same URI) constitute its corresponding *description*. For instance, the description of the resource `&r1` (lines 24-28) under the form of triples (`<subject, predicate, object>`) is given below:

```
<&r1 rdf:type Cubist>
<&r1 paints &r2>
<&r1 paints &r3>
<&r2 rdf:type Painting>
<&r3 rdf:type Painting>
<&r1 rdf:type Sculptor>
```

Moreover, triples can also be used to represent RDF schemas. For example, the description of the classes *Artist* and *Artifact* as well as that of the property *creates* (line 1-6) is represented by the following triples:

```
<Artist rdf:type rdfs:Class>
<Artifact rdf:type rdfs:Class>
<creates rdf:type rdf:Property>
<creates rdfs:domain Artist>
<creates rdfs:range Artifact>
```

An alternative RDF/S representation is based on *directed graphs with labels on nodes and edges*. This representation extends well-known semistructured data models [1.1] by taking into account the peculiarities of RDF/S: (meta)schema (or data) graph nodes are labeled with (meta)class names (or resource URIs) while edges defined between these nodes are labeled with property names. Classes and properties can also be related at the (meta)schema layer though additional *subsumption* edges while nodes at the various abstraction layers are connected thought *instantiation* edges. Figure 1.1 presents visually the graph representation of a cultural Portal catalog, a small part of which was represented previously in the RDF/XML syntax.

The upper part of Figure 1.1 refers to the metaschema layer employed by our application example and consists of the metaclasses of classes *RealWorldObject* and *WebResource* and the metaclass of properties *SchemaProperty*, whose instances are the properties *related* and *maxCardinality*. The middle part of Figure 1.1 refers to the schema layer of our application example and essentially comprises two schemas. The schema on the left defines basic cultural entities, such as *Artist* and *Artifact*, and related properties, such as *technique*, *creates* and *exhibited*. The schema on the right provides the concepts needed by the application to describe the resources according to the administrative information they contain, e.g., the title (*title*), the date of last modification (*last_modified*) and the size (*file_size*) of each resource. Both classes and properties are further refined, forming subsumption hierarchies. The lower part of Figure 1.1 represents the resources descriptions created according to these schemas. As previously stated, properties can be inherited (e.g., the subclass *Sculptor* also has the property *lname* and therefore this property can be used in the description of resource `&r12`), they can be attached more than once to a resource (e.g., the property *paints* to resource `&r1`), they are unordered (e.g., the properties *fname* and *lname* of `&r1` may occur in any order) and optional (e.g., the property *lname* is not used by the

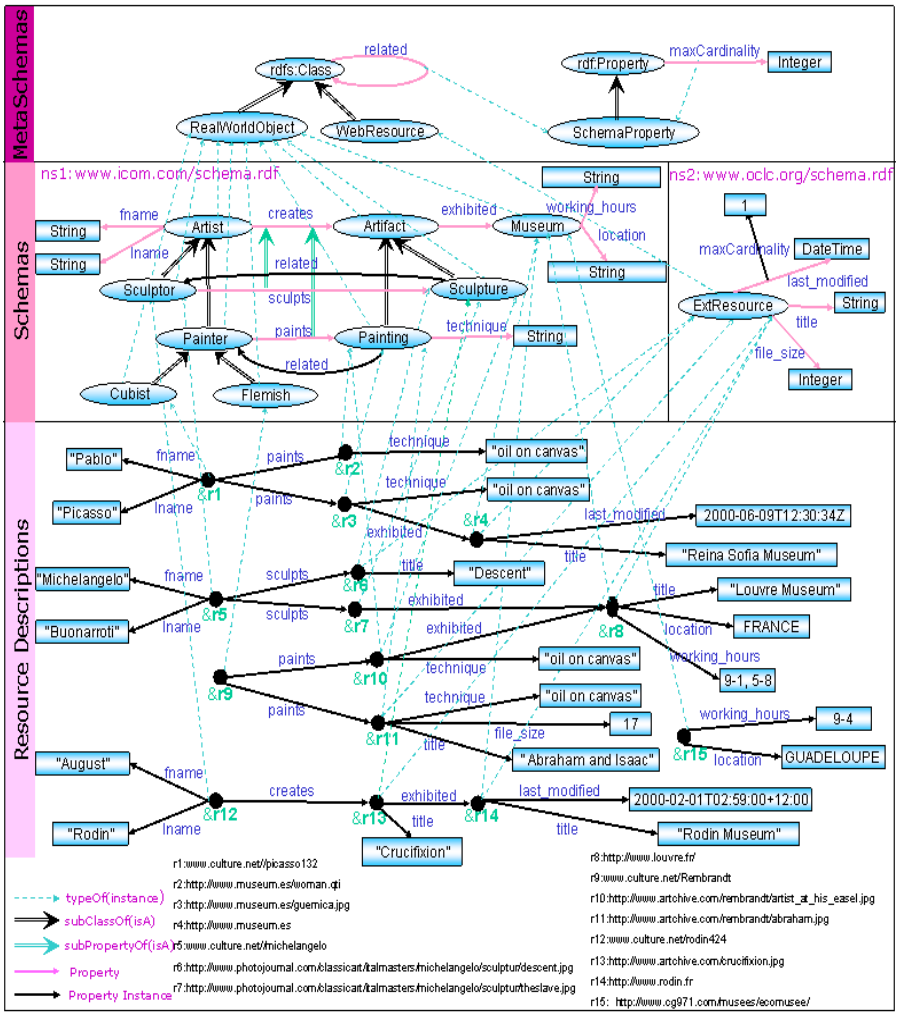


Fig. 1.1. An example cultural portal

resource `&r9`). Note that due to *multiple classification*, the resource `&r11` is classified under the classes *Painting* and *ExtResource* (not pairwise related with a subsumption relationship) and therefore `&r1` can be described using properties attributed to both classes.

Compared to the *triple*-based representation, the *graph*-based representation of RDF/S schema and resource descriptions provides a number of significant advantages. End-users are liberated from the arduous task of expressing joins over individual triples (i.e., edges) in order to navigate/filter complex description graphs. Developers, can more easily understand the semantics of the created and queried description graphs, especially when classes or prop-

erties are related through subsumption relationships and therefore additional triples can be inferred from the original descriptions. Finally, database engines have more optimization opportunities in terms of storage volumes and query execution times compared to a monolithic RDF/S representation by a flat and untyped triple-table. Readers are referred to [1.3, 1.15] for further details and experimental results on the efficient processing of RDF/S graphs.

1.2.2 A Formal Data Model for RDF

In this section we introduce a formal data model for *RQL* bridging and reconciling W3C RDF Model & Syntax with Schema specifications [1.22, 1.8], which is based on directed labeled graphs. Compared to the RDF/S specifications, the main contribution of the *RQL* model is the introduction of a semistructured type system for RDF schemas, as well as, the representation of RDF descriptions as atomic or complex data values. The connection between the two worlds is ensured by a type interpretation function, which (a) does not impose a strict typing on the descriptions (e.g., a resource may be liberally described using optional and repeated properties which are loosely-coupled with classes); (b) permits superimposed descriptions of the same resources (e.g., by classifying resources under multiple classes which are not necessarily related by subsumption relationships); and (c) allows for a flexible schema refinement (e.g., through specialization of both entity classes and properties).

RDF resource descriptions [1.22] are represented as *directed labeled graphs*, whose nodes are called *resources* (or *literal* and *container values*) and edges are called *properties*. RDFS schemas [1.8] are also represented as directed acyclic labelled graphs and essentially define vocabularies of labels for graph nodes, called *classes* (or *literal* and *container types*) and edges called *property types*.

More formally, we initially assume the existence of the following countably infinite and disjoint sets of symbols:
- $\mathcal{M} = \{m_1, m_2 \ldots\}$: metaclass names
- $\mathcal{C} = \{c_1, c_2 \ldots\}$: class names
- $\mathcal{P} = \{p_1, p_2 \ldots\}$: property names
- $\mathcal{U} = \{u_1, u_2 \ldots\}$: resource URIs
- $\mathcal{O} = \{o_1, o_2 \ldots\}$: container values
- $\mathcal{L}$: literals, strings, integers, dates, etc.

These sets of symbols are used to label the nodes and edges of an RDF/S graph (at the data, schema and metaschema abstraction layers). More specifically, $\mathcal{M}$ represents the set of metaclass names. Metaclasses can be distinguished into metaclasses of classes, denoted by $\mathcal{M}_c$, and metaclasses of properties, denoted by $\mathcal{M}_p$ ($\mathcal{M} = \mathcal{M}_c \cup \mathcal{M}_p$). The former includes also the default RDF/S name *rdfs:Class* and the latter the default RDF/S name *rdf:Property*, representing respectively the root of the subsumption hierarchy of metaclasses of classes and properties. The set $\mathcal{C}$ contains also the default RDF/S

name *rdfs:Resource* representing the root of the user-defined class hierarchy (schema layer).

Although not illustrated in Figure 1.1, RDF/S also supports structured values called *containers* (data layer). The set $\mathcal{P}$, apart from property names, includes also the arithmetic labels {1,2,3,...} used as property names for the members of container values. The set of all container values is denoted as $\mathcal{O}$. Each RDF/S container value can be uniquely identified by a URI and can be an instance of one and only one bulk type in Bt, namely *rdf:Bag*, *rdf:Seq* and *rdf:Alt*. Furthermore, the domain of every literal type t in Lt (e.g., string, integer, date, etc.) is denoted as `dom(t)`, while $\mathcal{L}$ represents the set $\cup_{t \in Lt} dom(t)$, i.e., the definition of the default RDF/S name *rdfs:Literal*. As a matter of fact, Lt represents the set of all XML basic datatypes that can be used by an RDF/S schema [1.25].

Each RDF schema uses a finite number of metaclass names $M \subseteq \mathcal{M}$, class names $C \subseteq \mathcal{C}$, and property names $P \subseteq \mathcal{P}$, as well as the sets of type names Lt and Bt. The property names are defined using metaclass, class, literal or container type names, such that for every $p \in P$, $domain(p) \in M \cup C$ and $range(p) \in M \cup C \cup Bt \cup Lt$ (`1.1`).

Definition 1 *An **RDF/S schema graph** RS is a six-tuple $RS = (V_S, E_S, \psi, \lambda, \prec, N)$, where $N = M \cup C \cup P \cup Bt \cup Lt$ and*

- *V_S is a set of nodes and E_S is a set of edges, where $V_S = M \cup C \cup Bt \cup Lt$ and $E_S = P$ (1.1)*
- *ψ is an incidence function $\psi : E_S \rightarrow V_S \times V_S$ (1.2)*
- *λ is a labeling function $\lambda : V_S \cup E_S \rightarrow 2^M$ (1.3)*
- *$\prec$ is a subsumption relation, such that:*
  - *`rdfs:Class` is the root of the hierarchy of metaclasses of classes*
  - *`rdf:Property` is the root of the hierarchy of metaclasses of properties*
  - *`rdfs:Resource` is the root of the class hierarchy (1.4)* □

The *incidence function* ψ represents the domain (*rdfs:domain*) and range (*rdfs:range*) of properties and imposes the restriction that the domain and range of a property must be unique (`1.2`). Using the example of Figure 1.1, the property edge *creates* connects the class nodes *Artist* and *Artifact*. The *labeling function* λ captures the *rdf:type* edges connecting the names of the schema layer with those of the metaschema (`1.3`). In particular, applied to the nodes of an RDF/S schema graph, λ returns the names of one or more metaclasses of classes, while applied on edges, it returns the names of one or more metaclasses of properties. For instance, the schema class *Artist* has an *rdf:type* edge to the metaclass of classes *RealWorldObject*. The incidence function ψ and the labeling function λ are *total* on the sets E_S and $V_S \cup E_S$ respectively. This fact does not exclude the case of nodes not related with a schema property edge. Furthermore, we assume that the node and edge labels of an RDF/S schema graph are unique, i.e., we adopt a *unique name assumption* in N (possibly using namespace URIs for disambiguation). (Meta)schema names are finally organized in subsump-

tion hierarchies through the relation "$\prec$", capturing the *rdfs:subClassOf* and *rdfs:subPropertyOf* edges (1.4).

In order to provide a crystal clear definition of the RDF/S data model semantics, we introduce the notion of a *valid RDF/S schema graph* imposing adequate restrictions on the formed RDF/S schema graphs.

Definition 2 *An RDF/S schema graph* $RS = (V_S, E_S, \psi, \lambda, \prec, N)$ *is* ***valid*** *if and only if:*

- *For the labeling function* λ*, it holds that:*
  - ◇ *if* $c \in C$ *and* $\lambda(c) = \{m_1, \ldots, m_n\}$*, for every* $i \in [1 \ldots n]$*,* $m_i \in M_c$ *(2.1)*
  - ◇ *if* $p \in P$ *and* $\lambda(p) = \{m_1, \ldots, m_n\}$*, for every* $i \in [1 \ldots n]$*,* $m_i \in M_p$ *(2.2)*
- *For the subsumption relation* $\prec$*, it holds that:*
  - ◇ *the relation* $\prec$ *is a strict partial order (irreflexive, antisymmetric and transitive relation)* [4] *(2.3)*
  - ◇ *if* $p_1, p_2 \in P$ *and* $p_1 \prec p_2$*, then* $domain(p_1) \preceq domain(p_2)$ *and* $range(p_1) \preceq range(p_2)$ *(2.4)*
  - ◇ *for every* $n \prec n', n, n' \in C$ *or* $n, n' \in P$ *or* $n, n' \in M_c$ *or* $n, n' \in M_p$ *(2.5)*

□

Condition 2.1 states that every class must be an instance only of metaclasses of classes. Condition 2.2 states that a property must be an instance only of metaclasses of properties. Condition 2.3 imposes that the subsumption relation is essentially an *acyclic*, *binary order relation*, so that RDF/S schema or metaschema hierarchies form a *directed acyclic graph* (DAG). As a matter of fact, RDF/S subsumption hierarchies are essentially semi-lattices. Condition 2.4 imposes that the domain and range of a subproperty must be subsumed by the domain and range of its super-properties. Condition 2.5 restricts the application of the subsumption relation between (meta)schema names of the same kind (e.g., hierarchies between metaclasses and classes are not allowed). The above constraints guarantee that the *union of two valid RDF schema graphs is always valid* w.r.t. the inclusion semantics of (meta)class and property subsumption.

Resource descriptions are defined using a finite set of resource URIs $U \subseteq \mathcal{U}$, literal $L \subseteq \mathcal{L}$ or container values $O \subseteq \mathcal{O}$ and property names P, such that every $p \in P$ emanates from a node in U and ends to a node in $U \cup L \cup O$ (3.1).

Definition 3 *An* ***RDF resource description*** *RD, instance of an RDF/S schema graph* $RS = (V_S, E_S, \psi, \lambda, \prec, N)$*, is a quintuple* $RD = (RS, V_D, E_D, \psi, \lambda)$*, such that:*

- V_D *is a set of nodes and* E_D *is a set of edges, where* $V_D = U \cup L \cup O$ *and* $E_D = P$ *(3.1)*
- ψ *is an incidence function* $\psi : E_D \rightarrow V_D \times V_D$ *(3.2)*

[4] A relation R is *irreflexive*, when it does not hold that iRi. In case of class/metaclass and property hierarchies, it means that i$\not\prec$i. A relation R is *antisymmetric*, when: if iRj, then it does not hold that jRi. A relation R is *transitive*, when: if iRj and jRk, then iRk. The symbol $\preceq$ extends $\prec$ with equality (thus, the reflexive property holds).

- *λ is a labeling function $\lambda: V_D \rightarrow 2^C \cup Lt \cup Bt$ (3.3)* □

The *incidence function* ψ represents the set of relationships and attributes attached to resources (3.2). The *labeling function* λ captures essentially the *rdf:type* edges, connecting the RDF data graph with an RDF/S schema graph (3.3). In particular, applied to resource nodes λ returns the names of one or more classes, while applied to value nodes it returns either a literal or a bulk type name. The incidence function ψ and the labeling function λ are *total* on the sets E_D and V_D respectively.

As in the case of RDF/S schemas, we introduce in the sequel the notion of a *valid RDF resource description.*

Definition 4 *An RDF resource description $RD = (RS, V_D, E_D, \psi, \lambda)$ is* ***valid*** *if and only if RS is a valid RDF/S schema and:*
- *for every node $n \in V_D$:*
  - ◇ *if $n \in U \Rightarrow \lambda(n) \subseteq C$ (4.1)*
  - ◇ *if $n \in L \Rightarrow \lambda(n) \in Lt$ (4.2)*
  - ◇ *if $n \in O \Rightarrow \lambda(n) \in Bt$ (4.3)*
- *for every $p \in E_D$ from node n to node n':*
  - ◇ $\exists c \in \lambda(n), c \preceq domain(p) \wedge \exists c' \in \lambda(n'), c' \preceq range(p)$ *(4.4)* □

Condition 4.1 states that a data resource is instance only of classes. Condition 4.2 (4.3) states that a literal (container) value is an instance of one and only one literal (bulk) type. Condition 4.4 imposes that a data resource (or literal, container value) to whom a property is applied (ends), must be an instance of the class (or type) constituting the domain (range) of the property. The above constraints guarantee that the *union of two valid RDF resource descriptions is always valid* w.r.t. the inclusion semantics of class and property subsumption.

1.2.3 A Type System for RDF

Schemas or types have been traditionally exploited in the database world by query languages, such as OQL [1.11], for several reasons:

Clear data interpretation: a type system provides an unambiguous *understanding of the nature of RDF/S data* returned by a query. For example, we can understand that a URI identifies uniquely a data resource and not a class or property.

Error detection and safety: due to typing rules, we can —on the one hand— *ensure the safety of operations* and —on the other hand— *check the validity of their compositions.* For instance, arithmetic operations on class names are meaningless.

Better Performance: a type system can provide valuable clues for designing a better storage for RDF/S graphs, while it can facilitate the efficient processing of queries (e.g., rewriting path expressions).

In order to introduce a type system for the RDF/S data model, a number of requirements must be taken into consideration. Firstly, in contrast to object-oriented schemas, RDF/S properties (i.e., attributes and relationships) are self-existent. For instance, one can query the property *creates* regardless of whether it emanates from resources that are instances of the class *Artist* or its subclass *Painter*. Secondly, due to the existence of the data-schema-metaschema abstraction layers, the instances of a type could be data of another type. For example, instances of the metaclass *rdf:Property* are schema properties like *creates*, while instances of this property are (binary) sequences. Lastly, container values may have heterogeneous contents e.g., literal or other container values, resource URIs or (meta)class and property names. The type system foreseen by RQL is given below:

$\tau = \tau_{M_c} \mid \tau_{M_p} \mid \tau_C \mid \tau_P[\tau,\tau] \mid \tau_U \mid \tau_L \mid \{\tau\} \mid [1:\tau_1, 2:\tau_2, \ldots, n:\tau_n] \mid (1:\tau_1 + 2:\tau_2 + \ldots + n:\tau_n)$

where τ_{M_c} is a `metaclass of classes` type, τ_{M_p} is a `metaclass of properties` type, τ_C is a `class` type, $\tau_P[\tau,\tau]$ is a `property` type, τ_U is the type of `resource URIs` (including the URIs of namespaces), τ_L is a `literal` type in Lt, {.} is a `bag` type, [.] is a `sequence` type and (.) is the `alternative` type. Alternatives in our model capture the semantics of union (or variant) types [1.10] and they are also ordered (i.e., integer labels play the role of union member markers). Since there exists a predefined ordering of labels for sequences and alternatives, labels can be omitted (for bags, labels are meaningless). Furthermore, no subtyping relation is defined in RDF/S. The set of all types one can construct from the (meta)class or property names, the resource URIs and the literal or container types is denoted as `T`.

The RQL type system provides all the components we need to capture collections with homogeneous and heterogeneous contents, as well as to uniformly interpret metaclasses, classes and properties. Thus, classes and metaclasses can be interpreted as unary relations (i.e., bags) of type $\{\tau_U\}$ (data layer) and $\{\tau\} : \tau \in \{\tau_C, \tau_P\}$ (schema layer) respectively, while properties can be interpreted (data layer) as binary relations (i.e., bag of sequences) of type $\{[\tau_U, \tau_U]\}$ for relationships and $\{[\tau_U, \tau_L]\}$ for atomic attributes. When the property range is a bulk type then the corresponding interpretation involves container values of an appropriate `bag` or `sequence` or `alternative` type. The notation $\tau_P[\tau,\tau]$ for property types indicates the exact type of its domain and range (first and second position in the sequence). Properties whose domain and range are metaclasses, are interpreted (schema layer) as $\{[\tau_1, \tau_2]\} : \tau_1, \tau_2 \in \{\tau_C, \tau_P\}$ depending on whether the domain (range) of the property is a metaclass of classes or of properties. Generally speaking, (meta)schema names in RDF/S play a dual role, both as *labels* and *containers* (disambiguation depends on the operation context). For instance, the name *Artist* refers to the schema graph node with the same label (class label), but also to the resource URIs which are direct and indirect instances of this class.

The instantiation of (meta)schema names with a finite set of resource URIs (or schema names) is captured by appropriate *population functions.*

Definition 5 *A **class population function**, $\pi_c : C \rightarrow 2^U$, assigns a finite set of resource URIs to a schema class such that:*
– *for every $c, c' \in C, c \preceq c', v \in \pi_c \Rightarrow v \notin \pi_{c'}$ (5.1)* □

In order to capture instantiation of metaclasses of classes and properties we also define the *population functions* $\pi_{M_c} : M_c \rightarrow 2^C$ and $\pi_{M_p} : M_p \rightarrow 2^P$. Note that the population functions are the inverse of the labeling functions λ employed at each abstraction layer (definitions 1 and 3). These functions are *partial*, due to the fact that there may be (meta)classes without population. Furthermore, since we can multiply classify resources (or class, property names) under several (meta)classes, π_c (or π_{M_c}, π_{M_p}) is *non-disjoint.* However, condition `5.1` imposes that a resource URI (or class, property name) appears only once in the extension of a (meta)class even though it can be classified more than once in its subclasses (i.e., it belongs to its "closest" class wrt the defined subsumption hierarchy). It should be stressed that, by default, we use an *extended* (meta)class interpretation, denoted by π^*, including both the proper instances of a (meta)class and the instances of its subclasses:

$$\pi^*(n) = \pi(n) \cup \{\pi(n') | n' \prec n\}$$

The set of all values one can construct from the class or property names, the resource URIs and the literal and container values using the *RQL* type system is denoted as `V` and the *interpretation* function $[\![.]\!]$ of *RQL* types is defined as follows:

Definition 6 *Given a population function π of (meta)schema names, the **interpretation function** $[\![.]\!]$ is defined as follows:*
– *literal types:* $[\![\tau_L]\!] = dom(\tau_L)$
– *resource types:* $[\![\tau_U]\!] = u \in U$
– *metaclass types:* $[\![\tau_m]\!] = \{v | v \in \pi^*_M(m)\}$
– *class types:* $[\![\tau_c]\!] = \{v | v \in \pi^*_c(c)\}$
– *property types:* $[\![\tau_p[\tau_1, \tau_2]]\!] = \{[v_1, v_2] | v_1 \in [\![\tau_1]\!], v_2 \in [\![\tau_2]\!]\} \cup \{[v'_1, v'_2] | \tau_{p'}[\tau'_1, \tau'_2], v'_1 \in [\![\tau'_1]\!], v'_2 \in [\![\tau'_2]\!], p' \prec p\}$
– *bag types:* $[\![\{\tau\}]\!] = \{\{v_1, \ldots, v_j\} | j > 0, \forall i \in [1..j], v_i \in [\![\tau]\!]\}$
– *sequence types:* $[\![[\tau]]\!] = \{[1 : v_1, 2 : v_2, \ldots, n : v_n] | n > 0, \forall i \in [1..n], v_i \in [\![\tau_i]\!]\}$
– *alternative types:* $[\![(1 : \tau_1 + 2 : \tau_2 + \ldots + n : \tau_n)]\!] = \{i : v_i \,|\, \forall i \in [1 \ldots n], v_i \in [\![\tau_i]\!]\}$ □

In order to ensure a *set-based* interpretation of *RQL* types, restriction (`5.1`) on (meta)class population is also applied to the interpretation of subproperties. In the rest of the chapter, we use the terms *class* and *property extent* to denote their corresponding interpretations.

On the whole, taking advantage of the *RQL* types `T` and values `V`, we introduce formally the notions of a *description schema* and *base.*

Definition 7 *A **description schema S** is a tuple $S = (RS, \sigma)$, where $RS = (V_S, E_S, \psi, \lambda, \prec, N)$ is a valid RDF/S schema graph and σ is a type function $\sigma : N \rightarrow T$* □

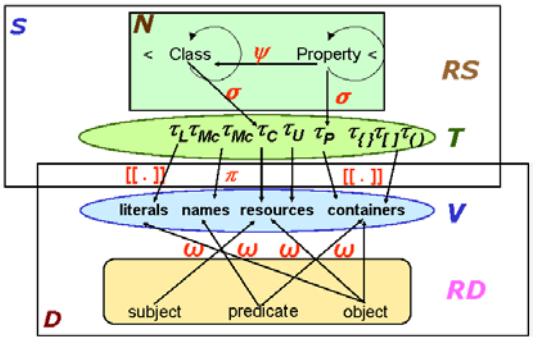


Fig. 1.2. The RQL formal data model

The *typing function* σ, which is *total* on N, relates (meta)class names with (meta)class types and property names with property types. In addition, it relates the basic XML Schema or RDF/S container type names with the *RQL* literal or bulk types.

Definition 8 *A* ***description base*** D*, instance of a description schema* $S = (RS, \sigma)$*, is a tuple* $D = (RD, \omega)$*, where* $RD = (RS, V_D, E_D, \psi, \lambda)$ *is a set of valid RDF resource descriptions and* ω *is a valuation function* $\omega : V_D \cup E_D \rightarrow V$*, such that:*

- *for every* $n \in V_D$, $\omega(n) \in [\![\sigma(\lambda(n))]\!]$ *(8.1)*
- *for every* $p \in E_D$*, from node* n *to node* n', $[\omega(n), \omega(n')] \in [\![\sigma(p)]\!]$ *(8.2)* □

The *valuation function* ω relates the nodes and edges of RDF resource descriptions with one of the values in V. Conditions `8.1` and `8.2` impose that the value of a node (edge) belongs to the interpretation of the type attached to the label of that node (edge). Finally, atomic nodes valuated with literals belong to the interpretation of concrete types like *string*, *integer*, *date*, etc.

In Figure 1.2 we summarize graphically the formal definitions introduced in this section. An `RDF schema graph RS` consists of a set of names `N`, connected through subsumption ($\prec$) and property edges (ψ). An `RDF resource description RD` is also a graph comprising a set of resource URIs and literal (or container) values which are connected through property edges. Both graphs can be represented using triples of the form `<subject, predicate, object>`. Then, when `RS` and `RD` satisfy appropriate validity constraints we are able to map the schema names `N` to a finite set of *RQL* types `T` (function σ) and the description triples to a finite set of *RQL* values `V` (function ω). `RS` and `T` constitute a `description schema S`, while `V` and `RD` constitute a `description base D`, which are connected through a type interpretation $[\![.]\!]$ mapping *RQL* types to values.

Compared to RDF/S [1.22, 1.8] and the recently proposed RDF Semantics [1.19], as well as *OWL* [1.18] and its predecessor *DAML+OIL* [1.16], the *RQL* data model and type system: (a) make a clear distinction between the different RDF/S abstraction layers (data, schema, metaschema), (b) enforce the constraint that *the domain and range of a property must always be defined and be unique*, and (c) forbid the existence of cycles in the subsumption hierarchies. The first constraint allows the unambiguous definition

of appropriate interpretation functions to pass from one layer to another. Regarding the domain and range of properties, permitting an optional declaration of multiple domains and ranges, as RDF Semantics [1.19] does, results in properties whose semantics are completely unclear and whose values may be both resources and literals. This freedom leads to semantic inconsistencies: resources should be uniquely identified by their URIs, while literals by their values. Moreover, this constraint ensures that specialized properties preserve set inclusion semantics of their domain and range. Finally, the introduction of cycles may considerably affect the semantics of already created RDF/S schemas and resource descriptions, especially when the subclass declarations are provided in many, different namespaces, while class equivalence relationships may still be expressed explicitly, using properties as *equivalentTo* — defined by *OWL* [1.18]— that have as domain and range metaclasses.

1.3 The RDF Query Language: RQL

The data model and type system previously presented enable the definition of a well-founded semantics of a declarative RDF/S query language involving recursion and functional composition. Exploiting its formal background, *RQL* constitutes a typed, declarative query language for uniformly navigating on RDF/S graphs at all abstraction layers.

In the next sections, we are going to illustrate *RQL*'s expressiveness by means of queries of increasing complexity, while exhibiting how this querying flexibility is accomplished due to the type system used (for an exhaustive comparison of *RQL* with other RDF Query languages, readers are referred to [1.24]). The example of Figure 1.1 is employed as a running example for all the *RQL* queries presented in this section. Readers can explore the expressive power of *RQL* with the *RQL* online demo[5], where one can experiment with queries and see the results either in RDF/XML syntax or in HTML produced after appropriate XSLT processing.

1.3.1 Basic Querying Functionality

Due to its functional nature, *RQL* can compose basic queries and iterators into complex queries. The basic *RQL* queries essentially constitute a simple browsing interface with minimal knowledge of the employed schema(s) for RDF description bases. For instance, in Knowledge Portals for each topic (i.e., class), one can navigate to its subtopics (i.e., subclasses) and eventually discover the resources (or their total number) which are directly classified under them. Similar needs are exhibited for the classification schemas used in E-Marketplace registries. To accommodate these needs, *RQL* provides appropriate functions for traversing recursively the class/property hierarchies defined in a (meta)schema, such as `subClassOf`/`subPropertyOf` and

[5] http://139.91.183.30:9090/RDF/RQL/

`superClassOf/superPropertyOf`. In order to retrieve the direct subclasses or superclasses of a (meta)class, these query expressions can be used with the operator "`^`" (respectively for properties). For example, the query

```
subClassOf^(Artist)
```

returns a bag with the class names *Painter* and *Sculptor*. These functions may also be used with a second integer parameter, in order to return, e.g., subclasses of *Artist* up to depth 2:

```
subClassOf(Artist, 2)
```

Furthermore, to retrieve root and leaf nodes of the subclass/subproperty hierarchies at the schema layer, *RQL* provides predefined functions with no arguments, namely `topclass` (`topproperty`) and `leafclass` (`leafproperty`). To query the metaschema these functions have overloaded signatures so they can be applied on metaclasses e.g.,

```
subClassOf(Class)
```

will return the metaclasses (of type τ_{M_c}) `RealWorldObject` and `WebResource`. In order to find the definition of a specific property one can use the functions `domain` and `range`. For instance,

```
domain(creates)
range(creates)
```

retrieve the domain and range of property *creates*, thus returning the class names *Artist* and *Artifact* respectively. On the other hand, `domain(maxCardinality)` will return the metaclass *Property*.

RQL's ability to functionally compose basic query expressions into more complex ones is ensured by the type system presented in Subsection 1.2.3. For instance,

If e is an expression of type class (or metaclass), then `subClassOf(e)` *(or* `superClassOf(e)`*) is a valid expression of type bag of classes (or metaclasses). Otherwise a type error is returned.*

This kind of restrictions and inferences is captured by the typing rules of basic data and schema queries illustrated in Tables 1.1 and 1.2. Each rule represents the drawing of a conclusion (the part below the horizontal line) on the basis of a premise (the part above the horizontal line); static type-checking is based on the validity of this premise for the corresponding *RQL* expression. For example, the expression `range(Artist)` will return at compile-time a type error since the function `range()` takes as operand only expressions (including names) of type property (rule "8" of Table 1.1).

More generally, since we are placed in a semistructured context, the whole RDF/S description graph can be viewed as a collection of nodes/edges. Then, schema nodes and edges can be queried as normal data using metaclass names, which serve essentially as entry-points to the corresponding graph (rule "10" of Table 1.2). Using, for instance, the core RDF metaclasses `Class` and `Property` (of type τ_M) as basic *RQL* queries, we obtain in our example the names of all classes (of type τ_C) and properties (of type τ_P) illustrated in Figure 1.1. Of course, user-defined metaclasses (e.g., `RealWorldObject` or `SchemaProperty`) can also be used as basic queries to retrieve the schema

Table 1.1. Schema Operations

Query Expression	Typing Rules	
`sub/superClassOf(1)`	$\dfrac{e:\tau, \tau \in \{\tau_C, \tau_M\},\ e' \in \{subClassOf, superClassOf\}}{e'(e) : \{\tau\}}$	(1)
`sub/superClassOf(2)`	$\dfrac{e_1 : \tau, \tau \in \{\tau_C, \tau_M\}, e_2 : integer,\ e' \in \{subClassOf, superClassOf\}}{e'(e_1, e_2) : \{\tau\}}$	(2)
`sub/superPropertyOf(1)`	$\dfrac{e:\tau_P[\tau_1, \tau_2],\ e' \in \{subPropertyOf, superPropertyOf\}}{e'(e) : \{\tau_P[\tau_1, \tau_2]\}}$	(3)
`sub/superPropertyOf(2)`	$\dfrac{e_1 : \tau_P[\tau_1, \tau_2], e_2 : integer,\ e' \in \{subPropertyOf, superPropertyOf\}}{e'(e_1, e_2) : \{\tau_P[\tau_1, \tau_2]\}}$	(4)
`top/leafclass`	$\dfrac{e \in \{topclass, leafclass\}}{e : \{\tau_C\}}$	(5)
`top/leafproperty`	$\dfrac{e \in \{topproperty, leafproperty\}}{e : \{\tau_P[\tau_1, \tau_2]\}}$	(6)
`domain`	$\dfrac{e : \tau_P[\tau_1, \tau_2], \tau_1 \in \{\tau_C, \tau_M\}}{domain(e) : \tau_1}$	(7)
`range`	$\dfrac{e : \tau_P[\tau_1, \tau_2], \tau_2 \in \{\tau_C, \tau_M, \tau_L\}}{range(e) : \tau_2}$	(8)
`typeof`	$\dfrac{e : \tau, (\tau = \tau_U \mid \tau \in \{\tau_C, \tau_P[\tau_1, \tau_2]\}}{typeof(e) : (\{\tau_C\} \mid \{\tau_M\})}$	(9)
`namespace`	$\dfrac{e : \tau, \tau \in \{\tau_C, \tau_M, \tau_P[\tau_1, \tau_2], \tau_L\}}{namespace(e) : \{\tau_U\}}$	(10)

classes or properties defined as their instances. In addition, to retrieve only the schema properties representing relationships or attributes of resources at the schema layer (e.g., all the properties of Figure 1.1 except *maxCardinality* and *related*), one can use the built-in *RQL* metaclass `DProperty`.

In the same style, we can access any RDF graph nodes and edges at the data layer, by just writing the appropriate schema names. For instance, the query:

```
Artist
```

returns a bag containing the URIs (i.e., $\{\tau_U\}$) `&r1`, `&r5`, `&r9` and `&r12`, since these resources belong to the extent of *Artist*. By considering properties as binary relations, the basic query:

```
creates
```

returns the bag of ordered pairs of resources (i.e., $\{[\tau_U, \tau_U\}$) belonging to the extended interpretation of *creates*.

In order to obtain only the proper instances of a class/property (i.e., only the nodes/edges labeled with the class/property name), one can use

Table 1.2. Data Operations

Query Expression	Typing Rules	
`count`	$\frac{e : \{\tau\}}{count(e) : integer}$	(1)
`min, max`	$\frac{e : \{\tau\}, \tau \in \{integer, float, date\}, e' \in \{min, max\}}{e'(e) : \tau}$	(2)
`avg, sum`	$\frac{e : \{\tau\}, \tau \in \{integer, float\}, e' \in \{avg, sum\}}{e'(e) : \tau}$	(3)
`bag`	$\frac{e_1 : \tau, e_2 : \tau, \ldots, e_n : \tau}{bag(e_1, e_2, \ldots, e_n) : \{\tau\}}$	(4)
`seq`	$\frac{e_1 : \tau_1, e_2 : \tau_2, \ldots, e_n : \tau_n}{seq(e_1, e_2, \ldots, e_n) : [\tau_1, \tau_2, \ldots, \tau_n]}$	(5)
`ith`	$\frac{e : [\tau_1, \tau_2, \ldots, \tau_n], i : integer, i \in [1..n]}{(e[i]) : \tau_i}$	(6)
`in`	$\frac{e : \tau, e' : \{\tau\}}{(e\ in\ e') : boolean}$	(7)
`set operations`	$\frac{e_1 : \{\tau\}, e_2 : \{\tau'\}, \tau = \tau', \theta = \{intersect, union, minus\}}{(e_1\ \theta\ e_2) : \{\tau\}}$	(8)
`comp`	$\frac{e : \tau, e' : \tau', \tau = \tau', \theta \in (=, !=, <, >, \leq, \geq, like)}{(e\ \theta\ e') : boolean}$	(9)
`^name`	$\frac{e : \tau, \tau \in \{\tau_C, \tau_M, \tau_P[\tau_1, \tau_2]\}}{\hat{}e : \{eval(\tau)\}}$	(10)

the restricting operator "^". In our example, the result of query `^Artist` is the empty bag, since no resource has been directly classified as instance of *Artist*. Due to the dual nature of RDF (meta)schema names both as labels and collections, we use in rule "10" of Table 1.2 an *eval* function, such that: $eval(\tau) = \tau'$ iff $\forall d \in [\![\tau]\!], d : \tau'$
This function returns the type of the instances of an RDF (meta)schema name, depending on its type; for example, instances of an RDF class are resources (of type τ_U). More precisely:

- $eval(\tau_M) = (\tau_C \mid \tau_P)$
- $eval(\tau_C) = \tau_U$
- $eval(\tau_P[\tau_1, \tau_2]) = [eval(\tau_1), eval(\tau_2)]$
- $eval(\tau_L) = \tau_L$
- $eval(\{\tau\}) = \{\tau\}$
- $eval([\tau_1, \tau_2, \ldots, \tau_n]) = [\tau_1, \tau_2, \ldots, \tau_n]$
- $eval((\tau_1 + \tau_2 + \ldots + \tau_n)) = \tau : \exists i[1 \ldots n], \tau = \tau_i$

For cases where several schemas are used at the same time, *RQL* provides the function `namespace`, in order to retrieve the (meta)schema namespace where a name is defined (rule "10", Table 1.1). For example, the query:

```
namespace(Artist)
```

returns the URI `http://www.icom.com/schema.rdf`. In cases where the same names are defined in different schemas, one can use the *RQL* `using namespace` clause, in order to resolve such naming conflicts explicitly, e.g.:

```
ns:Artist
USING NAMESPACE ns=&http://www.icom.com/schema.rdf#
```

Furthermore, using the *RQL* function `typeof` (rule "9", Table 1.1), one can find all classes (or metaclasses) under which a given resource (or class/property name) is classified. For example, the query:

```
typeof(&http://www.artchive.com/crucifixion.jpg)
```

returns a bag consisting of the class names *ExtResource* and *Sculpture*.

Apart from the basic querying functionality presented above, there are several other query facilities that add to the expressiveness of *RQL*. One of the additional querying facilities is the support of *set-based queries* and *container queries*. In particular, *RQL* supports common set operators (`union`, `intersect`, `minus`), which can be applied on collections of the same type (rule "8", Table 1.1). For example, the *RQL* expression:

```
Sculpture intersect ExtResource
```

returns a bag with the URIs of all resources which are classified under both *Sculpture* and *ExtResource*. According to our example, only the resources `&r6` and `&r13` are classified under both classes.

As we can observe from the above query, *RQL* also permits the manipulation of RDF container values. More precisely, we can explicitly construct Bags and Sequences using the basic *RQL* queries `bag` and `seq` (rules "4" and "5" , Table 1.2). To access a member of a Sequence we can use the operator "`[ ]`" with an appropriate position index (rule "6"). If the specified member element does not exist, a runtime error is returned. Furthermore, the Boolean operator `in` can be used to test membership in Bags (rule "7"). For example, the query:

```
seq(domain(creates), range(creates))[0]
```

returns the first element of the sequence, while

```
&www.culture.net/picasso132 in Painter
```

is true, since the resource *www.culture.net/picasso132* belongs to the extent of *Painter*.

For data filtering, *RQL* relies on standard Boolean predicates, such as `=`, `<`, `>` and `like` (for string pattern matching). All operators can be applied on literal values (i.e., strings, integers, reals, dates) or resource URIs (rule "9"). For example, the expression "*X*=*&www.museum.es*" is an equality condition between resource URIs. It should be stressed that this also covers comparisons between (meta)class or property names. In this case, the application of these predicates corresponds to a subsumption test. For instance, the condition "*Painter* < *Artist*" returns `true`, since the first operand is a subclass of the second. Disambiguation is performed in each case by examining the type of operands (e.g., literal values vs. URI equality, lexicographical vs. class ordering, etc.).

Table 1.3. Formal interpretation of the basic *RQL* path expressions

		Path Expression	Interpretation
Data Path	1.	$c\{X\}$	$\{v \mid v \in [\![\tau_c]\!]\}$
	2.	$\{X\}p\{Y\}$	$\{[v_1, v_2] \mid [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	3.	$\{X\}@P\{Y\}$	$\{[v_1, p, v_2] \mid p \in [\![\tau_{M_p}]\!], [v_1, v_2] \in [\![\hat{}\,\tau_p[\tau_1, \tau_2]]\!]\}$
	4.	$\$X\{Y\}$	$\{[c, v] \mid c \in [\![\tau_{M_c}]\!], v \in [\![\hat{}\,\tau_c]\!]\}$
Schema Path	5.	$Class\{X\}$	$\{c \mid c \in [\![\tau_{M_c}]\!]\}$
	6.	$\$X$	$\{c \mid c \in [\![\tau_{M_c}]\!]\}$
	7.	$Property\{P\}$	$\{p \mid p \in [\![\tau_{M_p}]\!]\}$
	8.	$@P$	$\{p \mid p \in [\![\tau_{M_p}]\!]\}$
	9.	$c\{\$C\}$	$\{c' \mid c' \in [\![\tau_{M_c}]\!], c' \preceq c\}$
	10.	$\{\$X\}p\{\$Y\}$	$\{[c_1, c_2] \mid c_1, c_2 \in [\![\tau_{M_c}]\!],$ $c_1 \preceq domain(p), c_2 \preceq range(p)\}$
	11.	$\$X\{\$Y\}$	$\{[c_1, c_2] \mid c_1, c_2 \in [\![\tau_{M_c}]\!], c_2 \preceq c_1\}$
	12.	$\{\$X\}@P\{\$Y\}$	$\{[c_1, p, c_2] \mid p \in [\![\tau_{M_P}]\!], c_1, c_2 \in [\![\tau_{M_c}]\!],$ $c_1 \preceq domain(p), c_2 \preceq range(p)\}$
Mixed Path	13.	$c\{X;\$C\}$	$\{[v, c'] \mid c' \in [\![\tau_{M_c}]\!], c' \preceq c, v \in [\![\hat{}\,\tau_{c'}]\!]\}$
	14.	$\{X;\$Z\}p\{Y;\$W\}$	$\{[v_1,c_1,v_2,c_2] \mid c_1,c_2 \in [\![\tau_{M_C}]\!], c_1 \preceq domain(p),$ $v_1 \in [\![\hat{}\,\tau_{c_1}]\!], c_2 \preceq range(p),$ $v_2 \in [\![\hat{}\,\tau_{c_2}]\!], [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	15.	$p\{Y;\$W\}$	$\{[v_2, c_2] \mid c_2 \in [\![\tau_{M_c}]\!], c_2 \preceq range(p),$ $v_2 \in [\![\hat{}\,\tau_{c_2}]\!], [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	16.	$\{X\}p\{Y;\$W\}$	$\{[v_1, v_2, c_2] \mid c_2 \in [\![\tau_{M_c}]\!], c_2 \preceq range(p),$ $v_2 \in [\![\hat{}\,\tau_{c_2}]\!], [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	17.	$\{X\}p\{\$W\}$	$\{[v_1, c_2] \mid c_2 \in [\![\tau_{M_c}]\!], c_2 \preceq range(p),$ $v_2 \in [\![\hat{}\,\tau_{c_2}]\!], [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	18.	$\{\$Z\}p\{Y;\$W\}$	$\{[c_1, v_2, c_2] \mid c_1, c_2 \in [\![\tau_{M_C}]\!], c_1 \preceq domain(p),$ $v_1 \in [\![\hat{}\,\tau_{c_1}]\!], c_2 \preceq range(p),$ $v_2 \in [\![\hat{}\,\tau_{c_2}]\!], [v_1, v_2] \in [\![\tau_p[\tau_1, \tau_2]]\!]\}$
	19.	$\{X;\$Z\}@P\{Y;\$W\}$	$\{[v_1, c_1, p, v_2, c_2] \mid p \in [\![\tau_{M_p}]\!], c_1, c_2 \in [\![\tau_{M_c}]\!],$ $c_1 \preceq domain(p), v_1 \in [\![\hat{}\,\tau_{c_1}]\!],$ $c_2 \preceq range(p), v_2 \in [\![\hat{}\,\tau_{c_2}]\!],$ $[v_1, v_2] \in [\![\hat{}\,\tau_p[\tau_1, \tau_2]]\!]\}$

Lastly, *RQL* is also equipped with a complete set of aggregate functions (`min`, `max`, `avg`, `sum` and `count`). For instance, we can inspect the cardinality of class extents (or of bags) using the `count` function, e.g.,:

```
count( Painting )
```

Note that the parameter of aggregate functions may be any query returning a collection of a proper type (rules "1"-"3").

1.3.2 RQL Filters

RQL supports SQL-like filters, which use *generalized path expressions* [1.2, 1.13, 1.14] with variables on nodes and edges to traverse RDF/S description graphs at arbitrary depths.

Thus, the `SELECT-FROM-WHERE` filters provide a powerful tool to iterate over collections with RDF data or schema information of any kind. The result

of an *RQL* filter is represented by an RDF Bag container value, on which we can define iterators using nested queries, while ordered tuples can be represented by RDF Sequences and, as we have already seen, they can be accessed through a position index (rule "6"). In particular, the `SELECT` clause defines —as usual— a projection over the variables whose values participate in the result. Moreover, we can use "`SELECT *`" to include in the result the values of all variables. This clause constructs an ordered tuple, whose arity depends on the number of projection variables. The `FROM` clause hosts the defined path expressions, which essentially define the part of the RDF/S graph that will participate in the evaluation of the query. In fact, a path expression consists of a series of steps. Each step represents movement in a particular direction by identifying node labels, and each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. These filtering conditions are declared at the (optional) `WHERE` clause. The result of each step is a list of nodes that serves as a starting point for the next step. Table 1.3 presents the formal interpretation of the basic kinds of *RQL* path expressions. Furthermore, we can use the (optional) clause `USING NAMESPACE` for the definition of namespace prefixes as explained previously.

RQL filters are used in a variety of contexts and application cases. The next sections illustrate how *RQL* exploits their presence to provide sofisticated querying functionality.

1.3.3 Schema Navigation

RQL extends the notion of generalized path expressions to entire class (or property) inheritance paths in order to implement *schema browsing or filtering* using appropriate conditions. This declarative query support for navigating through taxonomies of classes and properties is especially useful for real-scale applications, such as Portal catalogs, which employ large description schemas. Consider for instance the following query, where given a specific schema property we want to find all related schema classes:

```
SELECT $C1, $C2
FROM {$C1}creates{$C2}
```

The result of this query is a bag of type $\{[\tau_C, \tau_C]\}$ and is depicted in a tabular form as:

$C1	*$C2*
Artist	Artifact
Artist	Painting
Artist	Sculpture
Sculptor	Artifact
Sculptor	Painting
Sculptor	Sculpture
Painter	Artifact
Painter	Painting
Painter	Sculpture
Cubist	Artifact
Cubist	Painting
Cubist	Sculpture
Flemish	Artifact
Flemish	Painting
Flemish	Sculpture

As in the case of core schema and data operations, the validity of operations is ensured by means of appropriate type inference rules. Table 1.4

Table 1.4. Basic Filters

Path Expression	Typing Rules	
Data Paths	$\frac{e : \tau, \tau \in \{\tau_C, \tau_M\}, eval(e) : \tau'}{(\text{select } x \text{ from } e\{x\}) : \{\tau'\}, x : \tau'}$	(1)
	$\frac{e : \tau_P[\tau_1, \tau_2], eval(\tau_1) : \tau_1', eval(\tau_2) : \tau_2'}{(\text{select } x, y \text{ from } \{x\}e\{y\}) : \{[\tau_1', \tau_2']\}, x : \tau_1', y : \tau_2'}$	(2)
	$\frac{e : \{\tau\}}{(\text{select } x \text{ from } e\{x\}) : \{\tau\}, x : \tau}$	(3)
Schema Paths	$\frac{e : \tau_P[\tau_C, \tau_C]}{(\text{select } \$c_1, \$c_2 \text{ from } \{\$c_1\}e\{\$c_2\}) : \{[\tau_C, \tau_C]\}, \$c_1 : \tau_C, \$c_2 : \tau_C}$	(4)
	$\frac{e : \tau_P[\tau_1, \tau_2]}{(\text{select } \$\$c_1, \$\$c_2 \text{ from } \{\$\$c_1\}e\{\$\$c_2\}) : \{[\tau_1, \tau_2]\}, \$\$c_1 : \tau_1, \$\$c_2 : \tau_2}$	(5)
Mixed Paths	$\frac{e : \tau_P[\tau_C, \tau_C]}{(\text{select } \$c_1, x, \$c_2, y \text{ from } \{x; \$c_1\}e\{y; \$c_2\}) : \{[\tau_C, \tau_U, \tau_C, \tau_U]\}, \$c_1 : \tau_C, x : \tau_U, \$c_2 : \tau_C, y : \tau_U}$	(6)
	$\frac{e : \tau_P[\tau_1, \tau_2], eval(\tau_1) : \tau_1', eval(\tau_2) : \tau_2'}{(\text{select } \$\$c_1, x, \$\$c_2, y \text{ from } \{x; \$\$c_1\}e\{y; \$\$c_2\}) : \{[\tau_1, \tau_1', \tau_2, \tau_2']\}, \$\$c_1 : \tau_1, x : \tau_1', \$\$c_2 : \tau_2, y : \tau_2'}$	(7)

presents the typing rules used for *RQL* basic filters and rule "4" is applicable in this query. In particular, the `FROM` clause of this filter uses a basic *schema path expression* composed of the property name *creates* and two class variables *$C1* and *$C2* (Schema Path `10`, Table 1.3), with "{ }" used to introduce appropriate schema (or data) variables. In general, class variables are prefixed by "$" in order to be disambiguated syntactically from data variables and schema names.

Since RDF properties can be applied to any subclass of their domain and range (due to inclusion polymorphism), the path expression "{*$C1*}*creates*{*$C2*}" simply denotes that $C1 and $C2 iterate over `subClassOf(domain(creates))` and `subClassOf(range(creates))` respectively (including the hierarchy roots). We can observe that the above path expression essentially traverses the `rdfs:subClassOf` links in the schema graph. It should be stressed that such *RQL* path expressions can be composed not only of edge labels like *creates*, but also of node labels like *Artist*. For instance, "*Artist*{*$C*}" is a shortcut for `subClassOf(Artist){C}` (including the root class *Artist*) (Schema Path `9`, Table 1.3).

Let us now see how we can retrieve all related schema properties for a specific class:

```
SELECT  @P, range(@P)
FROM    {$C}@P
WHERE $C=Painter
```

The result of the above query contains all properties that may be applied on Painters, either because they are directly defined on class *Painter* or because they are inherited from a superclass of *Painter*. In the `FROM` clause of this query, we use another *schema path expression* composed of a class variable $C and a property variable @*P*. In general, property variables (of type τ_P) are prefixed by "@" and are implicitly range-restricted on the set of all data properties (i.e., *DProperty*). Then, for each valuation *p* of @*P*, the class variable $C ranges over `subClassOf(domain(p))`. The condition in the `WHERE` clause filters @*P* valuations to keep only those properties for which class *Painter* is equal to their domain (e.g., *paints*) or is a valid subclass of their domain (e.g., *creates*, *lname*, *fname*). In other words, the query is equivalent to the filtering condition "*domain(P)>=Painter*" evaluated over *DProperty* (i.e., `DProperty{P}`). We can observe that the above path expression traverses the `rdfs:domain` and `rdfs:range` links in conjunction to the `rdfs:subClassOf` links in the schema graph. Note that in the result, `range(P)` is of type union ($\tau_C + \tau_L$), since data properties may range to classes and literal types.

In cases where we only want to find the relationships that can be applied on *Painter*, i.e., only properties with class range, and iterate on their subclasses (i.e., get all possible range classes), we can use the query:

```
SELECT @P, $Y
FROM   {;Painter}@P{$Y}
```

In this case, "{;*Painter*}" simply denotes a filtering condition of schema nodes identified by the name *Painter* and taking into account the `rdfs:subClassOf` links. The symbol ";" is used as a syntactic way to disambiguate data variables from constant names, and determine the kind of the path (data vs. schema path) accordingly. Thus, in the expression "{;*Painter*}@*P*", the domain of @*P* is denoted to be *Painter* or any of its superclasses and it implies the filtering condition "*Painter>=domain(@P)*". The use of a schema variable restricts the result to the properties with class range. More specifically, it implies the condition "*range(@P) in Class*". In order to also include properties with literal or metaclass range, we use the prefix "$$" instead of "$". Furthermore, similar to the use of class names in path extremities, we can also use literal type names, where applicable, e.g., use the path expression "@*P*{;*string*}" to find all string-valued properties. Note, however, that such a restriction, if imposed on the domain of a property, would produce a compile-time type-error.

To illustrate the *RQL* schema querying capabilities combined with its functional semantics, consider the query which retrieves all information related to class *Painter* (superclasses as well as direct or inherited properties):

```
seq( Painter,
     superClassOf(Painter),
     ( SELECT @P, domain(@P), range(@P)
       FROM {;Painter}@P ) )
```

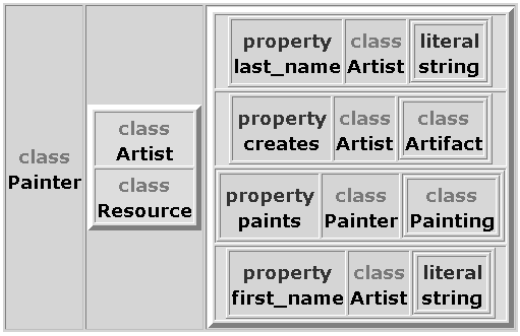


Fig. 1.3. HTML form of result

To collect all relevant information we explicitly construct a sequence with three elements. The first element is a constant (*Painter*) interpreted by the *RQL* type system as a class name (i.e., of type τ_C). The second element is a bag containing the names of the superclasses of *Painter* (i.e., of type $\{\tau_C\}$). The third element, which corresponds to the *RQL* filter, is a bag of sequences with three elements: the first of type property names (τ_P), the second of type class (τ_C), since a property domain is always a class, and the third of type union (i.e., Alternative) of class and literal type names. The result of the query in HTML form is presented in Figure 1.3.

We conclude this subsection, with a query illustrating how *RQL* schema paths can be composed to perform more complex schema navigation. For instance, the following query retrieves the properties that can be reached (in one step) from the range classes of property *creates*:

```
SELECT $Y, @P, range(@P)
FROM   creates{$Y}.@P
```

The result of this query is depicted in a tabular form as:

$\$Y$	$@P$	`range`($@P$)
Artifact	exhibited	Museum
Painting	exhibited	Museum
Sculpture	exhibited	Museum
Painting	technique	string
Sculpture	material	string

In the above query, the "." notation implies a join condition between the range classes of the property *creates* and the domain of $@P$ valuations: for each class name Y in the range of creates, we look for all properties whose domain is $\$Y$ or a superclass, i.e., "*$Y <= domain(@P) and $Y <= range(creates)*". This join condition enables us to follow properties which can be applied to range classes of *creates* (i.e., either because they are directly defined or because they are inherited) to any subclass of the range of *creates*. Schema path expressions may also be exclusively composed of property variables (with or without variables on domains and ranges). For instance, $@P.@Q$ will retrieve all two-step schema paths emanating from the subclasses

of the domain of @P and whose second part is either inherited from/defined on superclasses/subclasses of the domain of @Q. *RQL* schema

1.3.4 Data Navigation

The *RQL* generalized path expressions can also be used to *navigate/filter RDF description bases* without taking into account the (domain and range) restrictions implied by the properties in an RDF/S schema. This is quite useful since resources can be multiply classified and several properties coming from different class hierarchies may be used to describe the same resources. In this context, path expressions may be liberally composed from node and edge labels featuring both data or schema variables. In this case, the "**.**" notation is used to introduce appropriate join conditions between the left and the right part of the expression, depending on the kind of each path component and the specific operations defined by them (i.e., node vs. edge labels, data vs. schema variables). Consider, for example, the following query:

```
SELECT X, Y
FROM Museum{X}.last_modified{Y}
WHERE Y>=2000-01-01
```

In the `FROM` clause we use a *data path expression* with the class name *Museum* and the property name *last_modified*. The introduced data variables X and Y range respectively over the extent of class *Museum* (i.e., traversing the `rdf:type` links connecting schema and data graphs) and the *target* values of the extent of property *last_modified* (i.e., traversing properties in the RDF data graph). The "**.**" used to concatenate the two path components, implies a join condition between the *source* values of the extent of *last_modified* and X. Hence, this query is equivalent to the expression "*Museum*{X}, {Z}*last_modified*{Y} *where* $X = Z$". As we can see in Figure 1.1, the *last_modified* property has been defined with *domain* the class *ExtResource* but, due to multiple classification, X may be valuated with resources also labeled with any other class name (e.g., *Museum*, *Artifact*, etc.). The composition of the two paths is valid in terms of typing since both X and the implied Z are of type τ_U. Thus, due to multiple classification of nodes, we can query paths in a data graph that are not explicitly declared in the schema. For instance, "*creates.exhibited.title*" is not foreseen in any RDF schema, since the *domain* of the *title* property is the class *ExtResource* and not *Museum*. Yet, in our model X has the unique type τ_U, Y has type the literal type *date*, the implied join conditions are comparisons between values of type τ_U and the result of the above query is of type $\{[\tau_U, date]\}$. According to our example, this query returns the sites `www.museum.es` (`&r4`) with last modification date 2000-06-09 and `www.rodin.fr` (`&r14`) with date 2000-02-01.

More complex forms of navigation through RDF description bases are possible, using several data path expressions. For instance, to find the names

of Artists whose Artifacts are exhibited in museums, along with the related Museum titles, we issue the query:

```
SELECT V, R, Y, Z
FROM   {X}creates.exhibited{Y}.title{Z},
       {X}fname{V}, {X}lname{R}
```

In the `FROM` clause we use three data path expressions. Variable X (Y) ranges over the *source* (*target*) values of the *creates* (*exhibited*) property. Then, the reuse of variable X in the other two path expressions simply introduces implicit (equi-)joins between the extents of the properties *fname*/*lname* and *creates* on their *source* values. Since the *range* of property *exhibited* is the class *Museum* we do not need to further restrict the labels for the Y values in this query.

1.3.5 Combining Schema and Data Navigation

Up to now we have seen how we can query and navigate in schemas, as well as how we can query and navigate in description graphs **regardless** of the underlying schema(s). *RQL* allows the combination of schema and data filtering and navigation, through the use of *mixed path expressions* that enable us to *turn on* or *off* schema information during data filtering with the use of appropriate class and property variables. This functionality is illustrated in the following query, which finds the source and target values of properties emanating from ExtResources (similar to Mixed Path `19`, Table 1.3):

```
SELECT X, Y
FROM   {X;ExtResource}@P{Y}
```

The *mixed path expression* of the above query, features both data (X, Y) and schema variables on graph edges ($@P$). The notation "$\{X;ExtResource\}$" denotes a restriction on X to the resources that are (transitive) instances of class *ExtResource*. $@P$ is of type τ_P and is valuated to all properties having as a domain *ExtResource* or one of its superclasses. Finally, Y is range-restricted, for each successful binding of $@P$, to the corresponding target values. X is of type τ_U, while the type of Y is a union of all the range types of `ExtResource` properties. According to the schema of Figure 1.1, $@P$ is valuated to *file_size*, *title* and *last_modified*, while Y will be of type ($integer+string+date$). Note that this mixed path expression is not equivalent to the data path expression "$ExtResource\{X\}.@P\{Y\}$", which returns as a result not only the values of the properties having as a domain *ExtResource* but also those with domain any class under which instances of *ExtResource* are multiply classified (e.g., *exhibited*, *technique*). Note that, when using constant class or property names as the path's elements, (e.g., *Museum*, *last_modified*) path components are automatically considered as data paths, if no variables are defined on their extremities. On the other hand, paths containing property (@P) or class ($X) variables as their elements, are treated as schema paths (i.e., their domain or range is used to infer the implied condition).

The previous query introduced union type values in its result. The *RQL* type system is equipped with rules allowing us to infer appropriate union types, whenever it is required for query evaluation. Table 1.5 presents the typing rules applicable in cases of union type coercions. In these rules we use the `coerce` function, which returns the types of the given expression for which the corresponding operation is defined. The value of union type coercions is more obvious in the following query:

```
SELECT X, @P, Y
FROM {X}@P{Y}
WHERE Y >= 2000-01-01
```

In this query, Y is initially bound to a collection of type $(\tau_U + string + float + integer + date)$. Then, the condition in the `WHERE` clause imposes an additional implicit type condition: Y values should be of type *date* (rule "`Ucomp`" in Table1.5). As a consequence, the result of the query is of type $([\tau_U, \tau_P, date])$.

Using *RQL*'s mixed path expressions, one can start querying resources according to one schema, while discovering in the sequel how the same resources are described using another schema. To our knowledge, none of the existing query languages has the power of *RQL* path expressions. Consider, for example, the following query that finds related data and schema information of resources, whose URI matches "*www.museum.es*"

```
SELECT X, ( SELECT $W, ( SELECT @P, Y
                           FROM {X;$W}@P{Y} )
            FROM $W{X} )
FROM Resource{X}
WHERE X like ''www.museum.es''
```

In the above query, we are interested to discover, for each matching resource the classes under which it is classified and then for each class, the properties which are used along with their respective values. This grouping functionality is captured by the two nested queries in the `SELECT` clause of the external query. Note the use of string predicates such as `like` on resource URIs. Then for each successful valuation of X (of type τ_U), in the outer query, variable $\$W$ iterates over the classes (type τ_C) having X in their extent. Finally, for each successful valuation of X and $\$W$ in the inner query, variable $@P$ iterates over the properties which may have $\$W$ as domain and X as source value in their extent, by defining appropriate implicit filtering conditions on $@P$. According to the example of Figure 1.1, the type of Y is the union $(\tau_U + string + date)$. The final result of the query in HTML form is presented in Figure 1.4.

In cases where a grouped form of results is not desirable, we can easily generate a flat triple-based representation (i.e., subject, predicate, object) of resource descriptions. For instance, to retrieve the description of resources, excluding properties related to the class *ExtResource*, we issue the query:

Table 1.5. Union type coercions

Union Type Expression	Typing Rules	
`sub/super ClassOfU`	$\dfrac{e:(\tau_1+\tau_2+\cdots+\tau_n), \exists i \in [1\ldots n], \tau_i \in \{\tau_C, \tau_M\} \quad e' \in \{subClassOf, superClassOf\},}{e'(e):\{\Sigma_i(\tau_i)\}, coerce(e):\Sigma_i(\tau_i)}$	(1)
`inbagU`	$\dfrac{e:\tau, e':\{(\tau'_1+\tau'_2+\cdots+\tau'_n)\}, \exists i \in [1\ldots n], \tau'_i=\tau}{(e\ in\ e'):boolean, coerce(e'):\{\tau\}}$	(2)
`Uinbag`	$\dfrac{e:(\tau_1+\tau_2+\cdots+\tau_n), e':\{\tau'\}, \exists i \in [1\ldots n], \tau_i=\tau'}{(e\ in\ e'):boolean, coerce(e):\tau'}$	(3)
`UinbagU`	$\dfrac{e:(\tau_1+\tau_2+\cdots+\tau_n), e':\{(\tau'_1+\tau'_2+\cdots+\tau'_m)\}, \exists i\in[1\ldots n],\ j\in[1\ldots m],\ k\in[1\ldots l], \tau_i=\tau'_j(=\tau''_k)}{(e\ in\ e'):boolean, coerce(e):\Sigma_k(\tau''_k), coerce(e'):\{\Sigma_k(\tau''_k)\}}$	(4)
`Uintersect`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{\tau'\}, \exists i\in[1\ldots n], \tau_i=\tau'}{(e\ intersect\ e'):\{\tau'\}, coerce(e):\{\tau'\}}$	(5)
`UintersectU`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{(\tau'_1+\tau'_2+\cdots+\tau'_m)\}, \exists i\in[1\ldots n],\ j\in[1\ldots m],\ k\in[1\ldots l], \tau_i=\tau'_j(=\tau''_k)}{(e\ intersect\ e'):\{\Sigma_k(\tau''_k)\}, coerce(e):\{\Sigma_k(\tau''_k)\}, coerce(e'):\{\Sigma_k(\tau''_k)\}}$	(6)
`Uminus`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{\tau'\}, \exists i\in[1\ldots n], \tau_i=\tau'}{(e\ minus\ e'):\{(\tau_1+\tau_2+\cdots+\tau_n)\}}$	(7)
`minusU`	$\dfrac{e:\{\tau\}, e':\{(\tau'_1+\tau'_2+\cdots+\tau'_n)\}, \exists i\in[1\ldots n], \tau'_i=\tau}{(e\ minus\ e'):\{\tau\}, coerce(e'):\{\tau\}}$	(8)
`UminusU`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{(\tau'_1+\tau'_2+\cdots+\tau'_m)\} \quad \exists i\in[1\ldots n],\ j\in[1\ldots m],\ k\in[1\ldots m], \tau_i=\tau'_j(=\tau''_k)}{(e\ minus\ e'):\{(\tau_1+\tau_2+\cdots+\tau_n)\}}$	(9)
`Uunion`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{\tau'\}}{(e\ union\ e'):\{(\Sigma_i(\tau_i)+\tau')\}}$	(10)
`UunionU`	$\dfrac{e:\{(\tau_1+\tau_2+\cdots+\tau_n)\}, e':\{(\tau'_1+\tau'_2+\cdots+\tau'_m)\}}{(e\ union\ e'):\{(\Sigma_i(\tau_i)+\Sigma_j(\tau'_j))\}}$	(11)
`Ucomp`	$\dfrac{e:(\tau_1+\tau_2+\cdots+\tau_n), e':\tau', \theta\in\{=,!=,<,>,\leq,\geq,like\}, \exists i\in[1\ldots n], \tau_i=\tau'}{(e\ \theta\ e'):boolean, coerce(e):\tau'}$	(12)
`UcompU`	$\dfrac{e:(\tau_1+\tau_2+\cdots+\tau_n), e':(\tau'_1+\tau'_2+\cdots+\tau'_m), \exists i\in[1\ldots n],\ j\in[1\ldots m],\ k\in[1\ldots m], \tau_i=\tau'_j(=\tau''_k), \theta\in(=,!=,<,>,\leq,\geq,like)}{(e\ \theta\ e'):boolean, coerce(e):\Sigma_k(\tau''_k), coerce(e'):\Sigma_k(\tau''_k)}$	(13)

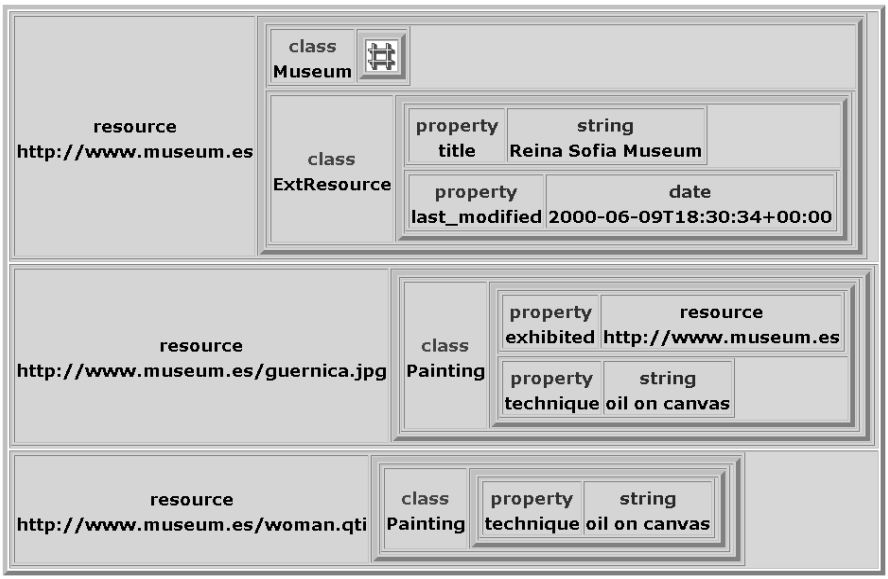


Fig. 1.4. HTML form of result

```
( ( SELECT X, @P, Y FROM {X}@P{Y} )
  union
  ( SELECT X, type, $W FROM $W{X} ) )
minus
( ( SELECT X, @P, Y FROM {X;ExtResource}@P{Y} )
  union
  ( SELECT X, type, ExtResource FROM ExtResource{X} ) )
```

In the above query, we essentially perform a set difference between the entire set of resource descriptions (i.e., the attributed properties and their values, as well as, the class instantiation properties) and the descriptions of resources which are instances of class *ExtResource*. Note that, since *RQL* query results are also typed collections, we are able to perform static type-checking when they are used (nested) in other queries (e.g. set operations, scans on nested queries). Regarding the typing of the two `union` query results, which contain union type values, rules 9 and 11 of Table 1.5 hold. First, the inferred type for the constants `type` and *ExtResource* (in the `SELECT` clause of the two union subqueries) is τ_P (i.e., a property name) and τ_C (i.e., class name). Second, variables Y and $\$W$ (in the `SELECT` clause of the first union) is of type $(\tau_U + string + float + integer + date)$ and τ_C. In this case, the union operation is performed between subqueries of different types.

1.4 Summary and Conclusions

In this chapter, we have presented a formal graph data model capturing the most salient features of RDF and a functional query language, *RQL*, for uniformly querying both RDF schema and resource descriptions. There currently exist two distinct implementations of *RQL*, one by ICS-FORTH (*139.91.183.30:9090/RDF/RQL*) and the other by Aidministrator (*sesame.aidministrator.nl/rql/*). The latter implementation however does not support the type system presented in this chapter. *RQL* is a generic tool used by several EU projects (i.e., *C-Web*, *MesMuses*, *Arion* and *OntoKnowledge*[6]) aiming at building, accessing and personalizing Community Knowledge Portals. In conclusion, expressiveness of querying and static type checking were the rationale behind our choice of a functional query language for *RDF*. It turns out from our experience that *RQL* provides as well sound foundations for the optimization of query evaluation. However this still remains a very hard issue. We started looking at this problem by providing adequate encoding schemes for class and property taxonomies in order to optimize some recursive queries [1.15].

References

1.1 Abiteboul, S., Suciu, D., Buneman, P. (1999): Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann Series in Data Management Systems

1.2 Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. (1997): The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, **1**(1):68–88

1.3 Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D. (2001): On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In Proceedings of the 4th International Workshop on the Web and Databases (WebDB'01) - In conjunction with ACM SIGMOD/PODS, Santa Barbara, CA

1.4 Berners-Lee, T., Fielding, R., Masinter, L. (1998): Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396. Available at: http://www.ietf.org/rfc/rfc2396.txt

1.5 Berners-Lee, T., Hendler, J., Lassila, O. (2001): The Semantic Web. Scientific American.
Available at: http://www.sciam.com/2001/0501issue/0501berners-lee.html

1.6 Bray, T., Hollander, D., Layman, A. (1999): Namespaces in XML. W3C Recommendation

1.7 Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. (2000): Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation

1.8 Brickley, D., Guha, R.V. (2000): Resource Description Framework Schema (RDF/S) Specification 1.0. W3C Candidate Recommendation

1.9 Brickley, D., Guha, R.V. (2002): RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft

[6] See cweb.inria.fr, cweb.inria.fr/mesmuses, dlforum.external.forth.gr:8080, www.ontoknowledge.org respectively

1.10 Cardelli, L. (1988): A Semantics of Multiple Inheritance. Information and Computation, **76**(2/3):138–164
1.11 Cattell, R.G.G, Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, C., Stanienda, T., Velez, F. (2000): The Object Database Standard: ODMG 3.0. Morgan Kaufmann Publishers
1.12 Chamberlin, D., Florescu, D., Robie, J., Simeon, J., Stefanescu, M. (2001): XQuery: A Query Language for XML. W3C Working Draft.
Available at: http://www.w3.org/TR/xquery/
1.13 Christophides, V., Abiteboul, S., Cluet, S., Scholl, M. (1994): From Structured Documents to Novel Query Facilities. In Proceedings of ACM SIGMOD Conference on Management of Data, Minneapolis, Minnesota (pp.313–324)
1.14 Christophides, V., Cluet, S., Moerkotte, G. (1996): Evaluating Queries with Generalized Path Expressions. In Proceedings of ACM SIGMOD Conference on Management of Data (pp.413–422)
1.15 Christophides V., Plexousakis D., Scholl M., Tourtounis S (2003): On Labeling Schemes for the Semantic Web, The Twelfth International World Wide Web Conference (WWW'03), Budapest, Hungary.
1.16 Connolly, D., Van Harmelen, F., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A. (2001): DAML+OIL (March 2001) Reference Description. W3C Note
1.17 Darwen, H. (Contributor), Date, C. (1997): Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL. Addison-Wesley
1.18 Dean, M., Connolly, D., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A., (2002): OWL Web Ontology Language 1.0 Reference. W3C Working Draft
1.19 Hayes, P., (2002): RDF Semantics. W3C Working Draft
1.20 Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D. (2001): Querying RDF Descriptions for Community Web Portals. In Proc. of BDA'2001 (the French Conference on Databases), Agadir, Morocco (pp. 133–144).
1.21 Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M. (2002): RQL: A Declarative Query Language for RDF. In Proc. of the Eleventh International World Wide Web Conference (WWW'02), Honolulu, Hawaii, USA (pp. 592–603).
1.22 Lassila, O., Swick, R. (1999): Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation
1.23 Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D. (2002): Benchmarking RDF Schemas for the Semantic Web. In: I. Horrocks and J. Hendler (Eds): ISWC 2002, LNCS 2342, pp. 132–146, Springer-Verlag
1.24 Magkanaraki, A., Karvounarakis, G., Tuan Anh, T., Christophides, V., Plexousakis, D. (2002): Ontology Storage and Querying. Technical Report 308, ICS-FORTH, Heraklion, Crete, Greece
1.25 Maloney, M., Malhotra, A. (2000): XML Schema part 2: Datatypes. W3C Candidate Recommendation

Author Index

Alexaki, Sophia 1

Christophides, Vassilis 1

Karvounarakis, Greg 1

Magkanaraki, Aimilia 1

Plexousakis, Dimitris 1

Scholl, Michel 1

Subject Index

directed labeled graphs 1, 5

functional composition 2, 17

interpretation function 12

namespace 4, 18

path expression 21
population function 12

RDF Schema Language 1, 4
– description schema, 13
– RDF/S schema graph, 8
resource 2
– class, 2
– metaclass, 2, 3
– token, 2
Resource Description Framework 1
– description base, 13
– RDF resource description, 10

triple representation 4
type system 11
typing rule 17, 22, 30