Dottorato di Ricerca in Informatica
Università di Bologna, Padova, Venezia

Searching and Retrieving in Content-based Repositories of Formal Mathematical Knowledge

Ferruccio Guidi

March 2003

Coordinatore:
Prof. Özalp Babaoğlu

Tutore:
Prof. Andrea Asperti

To Anna, whom I love so much.

Abstract

In this thesis, the author presents a query language for an RDF (Resource Description Framework) database and discusses its applications in the context of the HELM project (the Hypertextual Electronic Library of Mathematics).

This language aims at meeting the main requirements coming from the RDF community. in particular it includes: a human readable textual syntax and a machine-processable XML (Extensible Markup Language) syntax both for queries and for query results, a rigorously exposed formal semantics, a graph-oriented RDF data access model capable of exploring an entire RDF graph (including both RDF Models and RDF Schemata), a full set of Boolean operators to compose the query constraints, fully customizable and highly structured query results having a 4-dimensional geometry, some constructions taken from ordinary programming languages that simplify the formulation of complex queries.

The HELM project aims at integrating the modern tools for the automation of formal reasoning with the most recent electronic publishing technologies, in order create and maintain a hypertextual, distributed virtual library of formal mathematical knowledge. In the spirit of the Semantic Web, the documents of this library include RDF metadata describing their structure and content in a machine-understandable form.

Using the author's query engine, HELM exploits this information to implement some functionalities allowing the interactive and automatic retrieval of documents on the basis of content-aware requests that take into account the mathematical nature of these documents.

Acknowledgements

Many people helped me in different ways during these three years. My sincere gratitude goes especially to Prof. Andrea Asperti and to the whole HELM research group. In particular I want to thank Irene Schena and Claudio Sacerdoti Coen for their helpful discussions about the contents of this thesis. A special thank you goes to Prof. Fairouz Kamareddine for her precious suggestions on how to improve this document.
Finally, I'm grateful to Prof. Giovanni Sambin, who taught me constructive Mathematics and introduced me to Martin-Löf type theory: my first functional programming language. Some contents of this dissertation are clearly influenced by his teachings.

Contents

List of Figures

Preface

The HELM project (Hypertextual Electronic Library of Mathematics)[1] is meant to integrate the current tools for the automation of formal reasoning and the mechanization of mathematics (proof assistants and logical frameworks) with the most recent technologies for the development of Web applications and electronic publishing. The final aim is the development of a suitable technology for the creation and maintenance of a virtual, distributed, hypertextual library of formal mathematical knowledge.

In the spirit of the Semantic Web, the documents of the library are described by RDF (Resource Description Framework) metadata which provide information about their structure and content in a machine-understandable form. HELM aims at exploiting this information to provide its library with interactive and automated searching functionalities capable of retrieving documents on the basis of mathematically relevant requests.

The development of these functionalities involves the adoption of an RDF query engine working over HELM metadata base but current RDF query languages show the common drawback of failing to meet some important requirements as, for instance, the possibility of querying arbitrary RDF Schema graphs exploiting the information on RDF property hierarchies and the existence of a well defined semantics for the languages themselves.

The contribution of our[2] thesis is situated in the context of HELM and our objective is the development of an RDF query language that provides the main features requested by the RDF community and complies with the needs of HELM.

[1] See `<http://helm.cs.unibo.it/>`.

[2] Following a well-established tradition concerning the style of Italian Ph.D. dissertations and scientific papers in general, the author of this document will systematically refer to himself using *pluralis maiestatis*: i.e. first person plural instead of first person singular.

The peculiar aspects of our language concern query results that are highly structured and possess their own syntax, formally explained by a rigorous semantics.

We plan to use this language to test HELM metadata architecture, which is now under development, and to implement the searching functionalities needed to consult the library in the most profitable way both automatically and interactively.

This dissertation is structured as follows:

- Chapter 1 presents an introduction to formal mathematical knowledge (Section 1.1) focusing on the important role that the World Wide Web, and the Semantic Web in particular, can have in this field (Section 1.2). W3C contribution to the Semantic Web focuses on RDF metadata, which we describe in Section 1.3 with some applications. Finally we present RDF query languages focusing on the main requirements proposed by the RDF community (Subsection 1.4.1) and on current implementations and proposals (Subsection 1.4.2).
- Chapter 2 describes our query language focusing on the most relevant features (Subsection 2.1.1), on the main architectural issues (Subsection 2.1.2), on its operational semantics (Section 2.2) which concerns both queries and query results, and on the differences between the version presented here and the earlier versions (Section 2.3).
- Chapter 3 presents HELM at its present state focusing on the differences with related projects (Section 3.1.2), on the persistent contents of its library (Section 3.2) with particular regard to metadata, and on its software architecture (Section 3.3). HELM components are implemented in Caml: a functional language from the ML family.
- In Chapter 4 we describe how our query language is used inside HELM presenting the parsers and renderers for queries and query results (Subsection 4.1.1), the latest interpreter (Subsection 4.1.2), the query generator (Subsection 4.1.3) and the testing software (Subsection 4.1.4). The Overall performance of these components is analyzed in Section 4.2.
- Finally in Chapter 5 we draw some conclusions presenting planned future work.

The appendix contains a selection of the source code, written by us, implementing the software support for our language in the context of the HELM project.

Chapter 1

Introduction

1.1 Formal mathematical knowledge

1.1.1 Overview

In the last thirty years a specific class of software applications, including proof assistants and logical frameworks [He91, He93], was developed to address the problem of certifying the correctness of mathematical reasoning with quite notable results.

These tools manipulate mathematical notions (such as definitions, axioms, statements and proofs) encoded in a formal language (i.e. a language which is unambiguous both at the syntactic and at the semantic level) usually derived from a foundational theory such as type theory or set theory, offering a variety of features such as proof checking, step-by-step proof development assistance and automatic proof searching.

In most cases these mathematical notions are taken from papers, monographs, and textbooks where they are presented in a language which is necessarily not formal because a formal language, solving every ambiguity and imprecision, is often too constricting and verbose for humans. The problem of rephrasing informal notions in a formal language has been well put by Constable, Knoblock and Bates (1985):

> It is thus possible to translate the proof of any mathematical theorem into a completely formal proof. However, the prospect of actually doing this is quite daunting because an informal proof of modest length will expand to a formal one of prodigious size and will require in its production extreme care and detailed knowledge of the more or less arbitrary conventions of the particular

> formalism. These tedious details will in sheer number dominate the interesting mathematical ideas which are the raison d'être of the proof.

The first efforts in the direction of automated reasoning were based on top-down methods for proof searching: starting from the theorem to prove, the axioms are reached reconstructing the proof. The next step was the bottom-up approach: starting from the axioms level of the proof tree, an ever-expanding set of consequences is generated until the goal (i.e. the theorem to prove) is eventually reached. The most influential bottom-up method was resolution invented by Robinson (1965) [Rob65], based on unification.

At the end of the 70's the emphasis moved from the proof searching to proof assistance, proof checking and interactive theorem proving, which are closer to our present interest. The idea is not to prove difficult theorems automatically, but simply to assist humans in constructing a formal proof, or at the very least check its correctness.

A notable example was the AutoMath project led by N.G. De Bruijn, in which significant parts of mathematics were formalized: for example Bentham Jutting (1977) formalized the Landau's Grundlagen theory of the real number field [Jut94]. This system uses, as its formalization language, a version of the λC λ-calculus where the two abstraction operators (λ and Π) are identified [KBN99, KN96]. In this setting the Curry-Howard isomorphism defines a correspondence between types of the calculus and statements of the propositional logic, and between terms and proofs; finally the reduction of the calculus corresponds to cut elimination in proofs [GLT89].

The Mizar project [Rud92, TR93], which began a little later and is still quite vital today, on the contrary attempts, via a pseudo-natural representation of the mathematics, "not to depart too radically from the usual accepted practices of mathematics".

In 1972 Robin Milner introduced LCF (Logic for Computable Functions) a proof assistant whose formalization language is based on the Curry-Howard isomorphism combining interaction and automation.

In the last 20 years, several other systems have been developed, like [Coq][1], Lego [LP92], Nuprl[2], Isabelle[3], HOL [GM93] and PVS [ROSS99].

In particular Coq is based on the Calculus of (Co)Inductive Constructions (CIC) [CH88] and is one of the most advanced Proof Assistants. Nevertheless the great number

[1] See <http://coq.inria.fr/>.

[2] See <http://www.cs.cornell.edu/Info/Projects/NuPrl/>.

[3] See <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/>.

of offered functionalities (proof editing, proof checking, proof searching, proof extraction, program verification) have made the system very big and complex.

It is interesting to note that these tools are often conceived as monolithic architectures in the sense that they can be extended only using the utilities offered by the systems themselves, with the obvious limitations.

1.1.2 Availability and interoperability

The current tools for the mechanization of Mathematics allow to maintain repositories of certified mathematical notions but miss the aim of having standard accessible formal libraries because they are encoded into some application dependent format (usually a textual one in a specific tactical language, and a proof checked one in some internal concrete representation language). So the information is not directly available but can be accessed only through the functionalities offered by the specific application and this makes it suitable only for a limited number of tasks as well as too sensible to the evolution and the maintenance of the application itself.

For instance a huge amount of formalization work has been done since the 70's using several proof assistant prototypes, but this information can often be used only by a software application which is not maintained any more, and so is hopelessly lost.

Furthermore current encodings of formal Mathematics lack of satisfactory presentation formats, are often neither structured nor informative and the granularity or the meaning of expressions is not accessible. Namely the proof constructing procedures (sometimes called tactics) are context dependent and their level of detail rarely corresponds to the logical steps of proofs. Also pseudo-natural language descriptions are not good enough because they often loose the formal content of proofs.

Finally there is a foundational problem because different applications use different foundational languages and switching from one to another is not trivial at all.

The QED [Ano94] Manifesto of 1994 represents a proposal to draw the Theorem-proving communities together in a single project whose aim was to formalize all Mathematics using a foundational approach. Unfortunately QED didn't obtain a big success due to the difficulties of mutual encoding between not trivial logical systems.

The first and essential step in the direction of facing the interoperability problems between different applications is having a common standard representation layer for the

exchanged information. As a matter of fact, the standardization can not be forced at the level of logic, but can concern the content level and the communication technology.

A content-centric (and thus application independent) description of Mathematics in a highly structured and machine-understandable format has several advantages (see [Sch02]) as it allows: automatic elaboration and processing of content information, cut and paste of computationally meaningful expressions between two systems, automatic proof checking of published proofs, semantic search for mathematical concepts, indexing and classification by means of a rich and useful metadata level specifically oriented to Mathematics.

A technological standardization of the data representation layer allows to apply similar software tools (i.e. for searching, retrieving, displaying or authoring) to all mathematical documents regardless of their concrete nature which depends on the specific logical system.

Currently, the major achievements in this direction are obtained by the HELM project and by the MathWeb project.

The purpose of the HELM project[4] is the development of a suitable technology for the creation and maintenance of a virtual, distributed, hypertextual library of structured mathematical knowledge based on XML technology, trough the integration of the current proof assistants and logical frameworks with the most recent technologies for the development of Web applications and electronic publishing.

The purpose of the MathWeb project[5] is to support the development, use, and dissemination of an infrastructure for Web-supported mathematics. MathWeb is meant to provide software systems that connect a wide range of mathematical services by a common software bus as well as specific services supporting all aspects of doing Mathematics on the Web.

Both projects use XML (Extensible Markup Language) technology [XML] to store and exchange the mathematical information in fact XML, along with its applications, can be considered the best choice for storing, cataloguing, publishing, retrieving and processing information in a scalable, adaptive and extensible way.

One of the great strengths of XML is its flexibility in representing information coming from different sources, in fact XML is a metalanguage used to describe other languages.

[4]Hypertextual Electronic Library of Mathematics: `<http://helm.cs.unibo.it/>`.

[5]See `<http://www.mathweb.org>`.

An XML encoding has several benefits: it is standard and application independent thus it achieves reusability of data and it improves interoperability and communication among different applications, it is extensible and structured thus allowing complex operations and transformations to be done automatically. Moreover there is a rich set of standard tools for processing XML documents.

For these reasons XML is rapidly imposing as the main technological tool for all networking applications involving representation, manipulation and exchange of structured and distributed information.

1.2 Content-based Web technologies

1.2.1 Content-based technologies for serving Mathematics on the Web

As we said in the previous section, the existing proof assistants and logical frameworks are not suitable for the creation and the maintenance of large and easily accessible repositories of structured mathematical knowledge, whereas Web technologies surely achieve this purpose because they provide:

- Accessibility. Web documents are accessed and processed by many applications all over the world.
- Standardization. Mathematics can be served on the Web using standard formats that can capture both its structure and its semantics.

The main content-based technologies for serving Mathematics on the Web include three markup languages, MathML, OpenMath and OMDoc, which we recall here for reference.

The Mathematical Markup Language [MathML] is the instance of XML to be issued as a W3C[6] Standard to address the problem of encoding mathematical material for the Web, describing both its notational structure and its logical content with the aim of achieving interoperability and communication among software applications with different tasks.

MathML provides approximately 180 markup elements and joins a content markup language (C-MathML) to the most traditional presentational one (P-MathML). Presentation elements are conceived to express the two-dimensional layout and the structure of

[6]The World Wide Web Consortium: <http://www.w3c.org>.

mathematical notation. Content elements provide an explicit encoding of the underlying semantic structure of mathematical expressions.

C-MathML has been outfitted with enough XML elements to carry much of the meaning of Mathematics up to roughly the beginning of college level. Subjects covered to some extent by C-MathML are: arithmetic, algebra, logic and relations, calculus and vector calculus, set theory, sequences and series, elementary classical functions, statistics, linear algebra. Anyway C-MathML provides a generic extension mechanism to eventually define new content elements.

Content elements have a default presentation but mechanisms are provided to associate the two encodings in order to specify both the layout and the intended meaning of a mathematical expression.

The main interest of P-MathML is that it has been recommended by W3C as a standard for rendering mathematics on the Web and it is likely to be adopted and supported by most browsers.

The MathML encoding of mathematical expressions can be embedded in larger XML documents using the standard mechanism of Namespaces [XMLN].

The main drawback of MathML is that content description does not have a standard semantics, is restricted to common mathematical expressions and does not include other aspects of mathematical developments such as proofs and traditional structures like axioms, definitions, theorems, sections, theories and so on.

[OpenMath][7] is a standard for representing the semantic meaning of mathematical objects, allowing them to be exchanged between computer programs, stored in databases, or published on the Web.

Originally OpenMath was mainly focused on computer algebra systems, but this language is now attracting interest from other areas of scientific computation.

The architecture of OpenMath is based on the conversion between an XML representation of mathematical objects and its internal representation in a software application performed by an interface program called a Phrase Book. The translation relies on Content Dictionaries, hypothetically one for every theory, which contain the definitions of all entities of a particular logical system and their formal properties as well.

[7] See `<http://monet.nag.co.uk/openmath>`.

These definitions are given through semi-formal descriptions (namely a mixture of plain text containing natural language and OpenMath encoding expressing properties of the defined entity) so Content Dictionaries are machine-readable but not machine-understandable in a full sense.

Furthermore, as in C-MathML, proofs play a pretty marginal role in OpenMath.

OMDoc[8] [Koh00a, Koh00c, Koh00b] is a content-based markup format for communicating mathematics on the Internet, proposed in the context of the MathWeb project.

OMDoc is an extension of the OpenMath standard to the document level, supplying content markup and syntax to encode a structured theory hierarchy.

As mathematical documents (such as articles and interactive books) have a complex structure, this extension must solve two tasks: besides representing the semantics of documents, it must provide a standardized infrastructure of this as well.

As a consequence, OMDoc provides two sorts of markup. One markup addresses the microstructure of mathematical texts, which largely comprises the general pattern "definition, theorem, proof" but that can contain auxiliary items like explanatory text, cross-references, exercises or applets. The other markup addresses the macrostructure of mathematical texts in terms of mathematical theories, and focuses on the specification of formal theories of software and hardware behaviour.

OMDoc supports a proof format whose structural and formal elements are derived from the Proof Plan Data Structure (a hierarchical data structure developed for the Ωmega project[9], that represents a partial proof at different levels of abstraction, called proof plans), justified by tactic applications, based on a linearized natural deduction style, but also allows natural language representations at every level. This mixed representation enhances multimodal proof presentation and the accumulation of proof information in one structure.

OMDoc does not make any claim of semantic unification and allows to mix formal and informal content together, without a clear distinction between the two.

According to [Koh00a], OMDoc is a first attempt to standardize the notion of "module" (i.e. "theory" in a strong sense) but the issue of modules is still an open research field in the proof assistant community, so each application is likely to develop its own module

[8] Open Mathematical Documents: <http://www.mathweb.org/omdoc>.

[9] See <http://www.ags.uni-sb.de/~omega>.

system, easily incompatible with the OMDoc specification. Nevertheless, OMDoc has the important merit to provide an interesting and constructive starting point, emphasizing a major problem of current mathematical developments.

Finally we would like to recall that there are many other technologies for publishing mathematics on the Web (as for instance HTML pages with GIF images for equations, PDF documents, HTML pages with Design Science WebEQ or IBM Techexplorer components), but none of them is ideal [MT01, Ber98] mainly because they are presentation-oriented instead of content-oriented.

1.2.2 The Semantic Web

The World Wide Web was designed as a universal information space with the goal of being useful for both human communication and machines participation. One of the major obstacles to this has been the fact that most information on the Web is designed for human consumption rather than for data that can be processed automatically.

The Semantic Web [BHL01] is an extension of the Web in which machine-readable information is given a well defined meaning in a machine-understandable form intended for automated processing and thus allowing automated agents, sophisticated search engines and interoperable services.

The major challenge of the Semantic Web is to provide a language that can express both data and rules for reasoning about those data in a machine-understandable way.

The first step in this direction was the XML project, which began to address HTML limitations on structured documents, by selecting a yet extensible subset of SGML for use on the Web.

Now the W3C metadata (i.e. "data about data") activity, which is focused on the Resource Description Framework (RDF) [RDF, RDFS], addresses the completion of the following step: having a general metadata framework for modelling information on the Web (see [Mil98]).

Metadata provide the semantic information (i.e. descriptions about structure and content) about resources in a machine-understandable form and can be exploited for many applications that involve automated processing of data[10] like resource searching and indexing, intelligent software agents, digital signatures management and content rating.

[10] See <www.ukoln.ac.uk/interop-focus/presentations/cimi/taiwan/rdf/iap-html/sld001.htm>.

Some metadata are user-specified (like the keywords of a document) while other metadata may be generated automatically by a batch process (like the dependencies among documents) and represent redundant information which would be too expensive to compute on the fly.

Using metadata, search engines can "understand" the nature of resources much better, allowing more accurate searches. Software agents can use metadata to share and exchange knowledge, enhancing communication and interoperability between applications and Web communities. Digital signatures, the key to the "Web of Trust", will be encoded and transmitted as metadata.

Finally note that the distinction between data and metadata is not absolute but it is created by a particular application: many times the same resource will be interpreted in both ways simultaneously.

RDF provides a representation language for metadata [RDF] that extends the Platform for Internet Content Selection (PICS) [PICS] and describes the relations among resources in terms of named properties and values without any assumption about a particular application domain. RDF additionally provides a vocabulary description language (RDF Schema) [RDFS] to interpret metadata using specific vocabularies (i.e. class and property hierarchies) designed to encourage the exchange, reuse and extension of metadata packages defined by different resource description communities.

RDF is carefully designed to have the following characteristics:

- **Independence**: metadata should be described in a way that makes no assumptions about either their intended application domain or their intended semantics, in order to ensure that the descriptions can be reused in different domains.
- **Interchange**: metadata should be easy to transport and store.
- **Scalability**: huge sets of metadata should be easy to handle and process.

Moreover, RDF encourages the view of "metadata being data" by using XML (with Universal Resource Identifiers[11] [URI] and Namespaces) as its privileged technology to achieve these aims.

[11] A URI is structured string of characters whose intended meaning is associated by the applications managing it. A Uniform Resource Locator (URL) [URL] is a special kind of URI.

An RDF document contains assertions about resources (i.e. "objects") having properties with some values and this structure seems to be a natural way of describing the majority of the Web data processed by machines.

RDF assertions (or statements) can be encoded with XML tags describing triples of the form "resource-property-value" and each of these components can be denoted by a URI which ties it to a unique definition available on the Web.

Nevertheless it is still possible that different communities use different definitions of the same concept. A solution to this problem is provided by collections of information called ontologies [W3Cb] which define the terms used to describe and represent an area of knowledge in a computer-usable form.

Ontologies are usually expressed in a logic-based language, so that detailed, accurate, consistent, sound, and meaningful distinctions can be made among the defined terms.

Ontologies can prove very useful for a community as a way of structuring and defining the meaning of the metadata terms that are currently being collected and standardized. Furthermore they can be used to improve the accuracy of Web searches in which merging information from different communities is required.

The Semantic Web RDF-based data model [BHL01] is directly connected to the relational database model[12] as, in the context of RDF graph representation (see Section 1.3.1), a record corresponds to a node, a field name corresponds to a Property and a cell corresponds to a Value.

However relational database systems tend to have loosely enforced combination rules because a query can join tables by any columns with matching data type without any semantic check.

The Semantic Web is specifically designed to link different databases on the Web, allowing sophisticated operations across them to be performed by automated processing.

As we saw, current proof assistants and logical frameworks already offer huge repositories of fully structured, content oriented mathematical information, that can make a good test case for the Semantic Web. In this context, the mandatory precondition is producing and maintaining XML encodings of these repositories: that is what the HELM project is meant for (see Chapter 3).

[12] A relational database consists of tables made of rows, or records, which are sets of cells, or fields.

1.3 The Resource Description Framework

RDF provides a common framework for expressing information in such a way that it can be exchanged between applications without loss of meaning. The RDF suite of specifications focuses on the following issues:

- RDF model and syntax specification [RDF]: introduces a model for representing RDF metadata as well as a syntax for encoding and transporting them.
- RDF vocabulary description language [RDFS]: describes how to use RDF to express RDF vocabularies in general and defines a basic vocabulary for this purpose.
- RDF model theory [RDFMT]: specifies a model-theoretic semantics for RDF Model and Schema, and some basic results on entailment.

In this section we give a survey of the first two topics, leaving the third one aside. A conclusive subsection recalls some RDF applications as the Dublin Core Metadata Initiative and the EULER Metadata Set.

1.3.1 RDF models

The foundation of RDF is a model for representing named properties and property values which draws on well established principles from various data representation communities. In particular, the basic data model includes five object types: resources, properties, values, statements and containers.

- A **resource** is any entity that can be identified by a URI. In case of need, URI's are disambiguated using Namespaces.
- A **property** can be seen either as an attribute (i.e. an aspect or characteristic) of a resource, or as a binary relation between resources. Using RDF Schema, a property can be given a specific meaning defining its allowed values (i.e. the types of resources it can describe) and its relations with other properties (in the context of a property hierarchy). Properties can be given a name or can be anonymous.
- A **value** is a string or a primitive data type defined by XML that is used as a value or argument for a property of a resource (interpreting the property as an attribute or as a relation respectively). Values can contain unparsed XML markup or can be URI's and denote resources.

- A **statement** is a triple consisting of a resource r (the *subject*), a named property p (the *predicate*) and a value v (the *object*). This construction is used to model the assertion: "r has a property p whose value is v". In particular RDF predefines a property named *type*, whose values have to be resources, such that the statement $(r, type, r')$ models the assertion "r has type r'" (i.e. a type declaration).
- A **container** is a resource denoting a collection of values. There are three types of containers: bags (multisets), sequences (ordered multisets) and alternatives (lists of alternatives for the single value of a property). The membership relation between the container and the values it contains is defined by a set of predefined properties named *_1*, *_2*, *_3*, etc. In addition, a container has the *type* property appropriately set to one of these values (formally resources): *Bag*, *Seq* or *Alt*. Any property of a container describes the container itself and does not apply to its members. Nevertheless RDF provides some primitives to make statements that distribute over each member of a container.

RDF supports *reification* which, in this context, is the possibility of representing a statement with a resource, with the purpose of making statements about it (i.e. higher order statements). The reification of a statement (r, p, v) is a resource for which the *type* property has the value *Statement* (that formally is a resource) and for which the three properties *subject*, *predicate*, *object* are set to r, p, v respectively.

RDF provides three representations of the RDF data model:

- **RDF triples**: a collection of statements described in triples.
- **RDF Graph**: the RDF directed labelled graphs are a syntax-neutral way of representing RDF statements. In these diagrams a named property, the predicate, corresponds to an arc drown from the subject to the object.
- **RDF Syntax**: RDF uses XML encoding with Namespaces as its interchange syntax and in this context the predefined resources and properties should be prefixed by the "RDF:" Namespace. The syntax comes in two formats: the serialization format gives access to the full capabilities of the data model, while the abbreviated format includes additional features that provide a more compact representation form of a subset of the data model. RDF interpreters are expected to implement both formats, so metadata authors are free to mix them at will.

As RDF properties represent relationships between resources, the RDF data model can resemble an entity-relationship (ER) diagram. However RDF is more general than that as relationships are first class objects (in the sense that they are entities themselves). Furthermore, the set of relationships an entity can have is not defined when the entity is declared, as with object-oriented systems often used to implement ER models, and can be stored apart from the entity.

1.3.2 RDF Schema

The RDF vocabulary description language (RDF Schema or RDFS) provides means for describing properties and their relationships with other resources, and is specified in terms of the basic RDF information model (i.e. a graph structure describing resources and properties) using a basic hierarchical class system. In this context a class is a set of RDF values and is itself a resource.

in particular the RDF Schema specification includes a predefined RDF vocabulary for describing the meaningful use of properties and classes in RDF data. Therefore the constraints imposed by an RDF Schema have a semantic nature, and are not syntactical as those imposed by an XML Document Type Definitions (DTD) [XML] or an XML Schema [XMLS].

RDFS class system is similar to the type systems of object-oriented (OO) programming languages but differs from those in that properties are defined in terms of the classes to which they apply, whereas OO approach is to define a class in terms of the properties its instances may have.

For this purpose, RDFS defines two properties named *domain* and *range* which apply to properties themselves describing their domain (class of the subjects) and range (class of the objects). Note that Namespaces can be used in the URI's denoting domains and ranges so these classes can be inherited from separate vocabularies.

Other important properties predefined by RDF Schema are:

- *isDefinedBy*: the Namespace of a resource;
- *comment*: a human-readable description of a resource;
- *label*: a human-readable description of a resource name;
- *seeAlso*: a resource providing information about the subject resource;

- *subClassOf*: this allows to build class hierarchies;
- *subPropertyOf*: this allows to build property hierarchies.

Note that *subClassOf* and *subPropertyOf* are defined transitive and that a subproperty inherits the domain and range of ancestors.

RDF Schema also predefines the following main classes:

- *Class*: the class of classes[13];
- *Resource*: the class of resources;
- *Literal*: the class of RDF literals (i.e. values);
- *Container*: the class of RDF containers.

As for RDF Model, also RDF Schema uses XML encoding with Namespaces as its interchange syntax and the predefined classes and properties mentioned above should be prefixed by the "rdfs:" or "RDFS:" Namespace.

1.3.3 Some applications of RDF

The Dublin Core Metadata Initiative[14] is an open forum engaged in the development of interoperable on-line metadata standards that support a broad range of purposes and business models. The Dublin Core Element Set [DC] (DCES) contains 15 elements believed to be broadly applicable for the description of Web resources across disciplines and languages, and is considered a world-wide standard for its capability to qualify descriptions with domain-specific information.

The DCES elements can be represented in many syntax formats and RDF is one of these. In particular the DCES RDF Schema ("dc" Namespace) declares the following properties (one for each element).

- **title**: a name given to the resource.
- **creator**: an entity primarily responsible for making the content of the resource.
- **subject**: the topic of the content of the resource.
- **description**: an account of the content of the resource.

[13]This definition does not lead to inconsistency because RDFS classes need not to be proper classes.
[14]See `<http://dublincore.org/>`.

- **publisher**: an entity responsible for making the resource available.
- **contributor**: an entity responsible for making contributions to the content of the resource.
- **date**: a date (in YYYY-MM-DD format) associated with an event in the life cycle of the resource.
- **type**: a description of the nature of the content of the resource.
- **format**: a description of the physical or digital manifestation of the resource.
- **identifier**: an unambiguous reference to the resource within a given context.
- **source**: a reference to a resource from which the resource is derived.
- **language**: the language of the intellectual content of the resource[15].
- **relation**: a reference to a related resource.
- **coverage**: the extent or scope of the content of the resource.
- **rights**: the information about rights held in and over the resource.

Additional qualifier elements are provided in two categories:

- **Element Refinement**: these qualifiers make the meaning of an element more specific in the sense that a refined element shares the meaning of the unqualified element with a more restricted scope.
- **Encoding Scheme**: these qualifiers identify schemes for interpreting an element value. Such schemes include controlled vocabularies and formal notions or parsing rules.

EULER[16] is a project funded by the European Commission in April 1998 to integrate different mathematical resources available on the Web into a single electronic library for searching purposes. In particular, a common user interface (the EULER service) allows an homogeneous access to the following integrated information types: scientific literature databases, library catalogues (OPAC), electronic journals from academic publisher, archives of preprints and grey literature, indexes of mathematical Internet resources.

[15] See RFC 1766: <http://www.ietf.org/rfc/rfc1766.txt>.

[16] European Libraries and Electronic Resources in mathematical sciences: <http://www.emis.de/projects/EULER>.

The integration approach makes use of common resource descriptions based on an extension of the DCES and accesses those descriptions via the Z39.50 protocol[17]. Technically, providers produce DCES metadata for their resources and offer them as distributed databases that are located at the providers' sites. The central EULER engine queries these databases in parallel via a common Z39.50 profile and performs result set merging and presentation formatting.

1.4 Query languages for RDF metadata

In the previous section we saw that from the logical standpoint RDF metadata can be presented as a set of "subject-predicate-object" triples (statements) and querying this structure means searching and retrieving the subsets of statements satisfying a given pool of conditions (to be specified in the query).

An RDF query language is a syntax for expressing these queries, which is usually interpreted by a query engine responsible for retrieving the query results.

1.4.1 Main requirements

Some of the main parameters for the evaluation of such a language and engine appear to be the following [GS03, Bee98, DBSA98][18]:

1. It should be possible to query an **arbitrary RDF graph** which includes information coming from both RDF Models and Schemata in fact the possibility of inferring from Schemata allows to perform queries with a deeper semantic content. In particular the language should provide facilities for denoting classes and properties by:

   - hierarchical constraints based on *rdfs:subClassOf* and *rdfs:subPropertyOf*;
   - traversal of compound values of properties.

   At the same time the engine should exploit:

   - the transitivity of RDFS hierarchical structure to infer relationships between classes and properties that are not explicitly stated with *rdfs:subClassOf* and *rdfs:subPropertyOf*;

[17] See `<http://lcweb.loc.gov/z3950/agency/1995doce.html>`.

[18] See also `<http://www.w3.org/2001/11/13-RDF-Query-Rules/>`.

- the classification of resources based on their properties to infer for instance that a resource having a particular property is an instance of a particular class;
- the graph structure of Models and Schemata to infer inverse relationships.

2. The language should provide **operations on literals** such as comparisons and should allow **disjunctive Boolean conditions** in queries. Moreover it should be possible to select URI's and literals by **patterns** such as wild cards or regular expressions.
3. The language (and probably the underlying data storage facility) should provide access to the **history** of the queried data.
4. The engine should be able to retrieve **exhaustive solutions** of queries. An exhaustive solution may be retrieved all at once or with a "first-next" strategy, i.e.: the caller asks repeatedly for the next solution.
5. The language should allow to **customize the query results** by specifying what part of an exhaustive solution should be returned to the caller. In particular it should be possible to hide some details of the solution.
6. As the results of a query may be interpreted differently depending on the source of the data, the language should provide **data source identification**, i.e. provide the sources of the solutions.
7. The language should have a well conceived and **well defined semantics** including a precise definition of the underlying data model. In particular a single general semantic form of query resembling the **"select-from-where" pattern** is expected to satisfy all query requirements. Furthermore the language should be defined in terms of the **abstract RDF model**, while RDF syntactic representation, say its XML syntax, should be a secondary concern.
8. **Different syntaxes** might be used as appropriate for different purposes. In particular an XML syntax and a textual SQL/OQL-like syntax would be appreciated (a conventional SQL oriented syntax may be easier to understand for humans than an equivalent XML representation).
9. As querying RDF is only a subtask for more advanced techniques on the Web, the engine should be easily **usable in the Web environment** and integrable with other software components.

Notice that parameters 1, 2, 3 concern searching, 4, 5, 6 concern retrieving while 7, 8, 9 are presentational.

It is important to stress that, even if encoded in XML, RDF metadata requires a dedicated query language since a semi-structured query language like XQuery [XQuery] fails to capture the semantics of RDF. This is basically because the RDF modelling primitives are substantially different from those defined in object or relational database models. In particular classes do not define objects or relation types because an instance of a class is just a resource without any value or state. Furthermore, resources may belong to different classes not necessarily related by specialization, therefore they may have different properties while there may be no class on which the union of these properties is defined. Finally RDF properties may be refined and this concept is not present, say, in XQuery.

1.4.2 Proposals and implementations

The rdfDB Query Language seems to be the first query language focused on RDF metadata ad is the basis of Algae (see below). It was written by R.V. Guha to work with the rdfDB System[19]. This language provides support for RDF Schemata and for some basic forms of inferring. Constraint management is very limited.

[Algae][20] is another early SQL-like language that uses an S-expression syntax to describe a graph with variables in some node positions which is matched against an RDF graph. Algae was implemented by E. Prud'hommeaux in Perl and can be used with an SQL database holding RDF statements as triples. The database engine must implement the *statementsMatching* method which is called for each constraint of the algae query. Query results are returned in a 2-dimensional table. Currently Algae powers the Annotea[21] annotations system[22] as well as other W3C software.

SquishQL[23] is one of the many query languages syntactically similar to rdfDB QL. The query engine is implemented in Java and connects to the underlying database using JDBC technology.

[19] See <http://guha.com/rdfdb/>.
[20] See also <http://www.w3.org/2001/Talks/0505-perl-RDF-lib/Overview.html>.
[21] See <http://www.w3.org/2001/Annotea/>.
[22] See <http://www.w3.org/2001/Annotea/User/Protocol.html>.
[23] See <http://swordfish.rdfweb.org/rdfquery/>.

The DAML+OIL Query Language Proposal[24] rises in the context of the DARPA DAML project[25] and describes a language with XML syntax only.

RDF Query[26] is an RDF vocabulary (whose Namespace is "rdfq") to express queries on RDF models (i.e. no inferences from RDF Schema). A query operates on a source container of resources and returns a result container of resources (which is always a subset of the former) that can be the source for another query. The resources in the result container may have fewer properties than in the source and this feature, known as projection, allows to build different views of the source. The language supports user-defined aggregate functions on containers and some container composition operators such as *union*, *intersection* and *difference* which apply if the resources in the two operands have the same properties (same by name, not by value) so an aliasing mechanism is provided to change the property names. The duplicate elimination policy for these operators is not specified. There is a syntactic support for sorting the resources in a container. Queries can be universal and existential, Boolean conditions are very primitive but new operators can be easily added extending the markup.

In the above languages constraints can not be composed disjunctively and there is no syntactic support for hierarchical constraints inferred from RDF Schemata (parameters 1 and 2 stated in the previous section).

RDFQL is the query language implemented by G. Chappell for the RDF Gateway JDBC Driver[27] The query engine can understand and process inference rules and the language provides a full set of functions on strings, numbers, dates, URI's/URL's, lists and encryption plus all standard Boolean operators to combine constraints. Nevertheless it does not have a syntactic support for hierarchical constraints.

RDQL[28] is a Java implementation of an SQL-like query language for RDF derived from SquishQL. It treats RDF as data and provides query with triple patterns and constraints over a single RDF model. The language provides a full set of Boolean and arithmetic operators to be used in constraints but does not support transitive closures (i.e. hierarchical constraints). Its purpose, as stated on the RDQL Web site, is as a model-level access

[24] See <http://www.daml.org/listarchive/joint-committee/0572.html>.
[25] See <http://www.ai.sri.com/daml/notes/HW2/SemanticWeb/paper.html>.
[26] See <http://www1.coe.neu.edu/ srayavar/W3CQL/ql.htm>.
[27] See <http://www.intellidimension.com/default.asp>.
[28] See <http://www.hpl.hp.com/semweb/rdql.html>.

mechanism that is higher level than an RDF API.

RQL is the query language used by the ICS-FORTH RDFSuite[29] which includes a Validating RDF Parser, an RDF Schema specific database and The query engine. These tools aim at a uniform management of RDF Schemata and resource descriptions, in fact RQL supports transitive closures on RDF subclasses and subproperties at the syntactic level. RQL follows a functional approach (like OQL[30]) and is based on a formal graph model (as opposed to other triple-based RDF query languages) that captures the RDF modelling primitives and allows the interpretation of superimposed resource descriptions by means of one or more Schemata. RQL is implemented on top of a persistent RDF Store that exploits available RDF schema information in order to efficiently load and query resource descriptions in an Object-Relational DBMS (SQL3). Notice that query results are retrieved in 2-dimensional tables. RQL is currently used by the Sesame system[31].

RuleML is part of the Rule Markup Initiative[32] which aims at defining a shared Rule Markup Language allowing both bottom-up and top-down rules in XML for deduction, rewriting, and further inferential-transformational tasks. The current version of the language (XML/RDF-combining RuleML 0.8) does not support Boolean disjunction.

XDD[33] is a language that enables representation of the semantics of a Web resource. It is founded on a theoretical basis upon which representation, computation and reasoning about XML data can be carried out in a uniform and succinct manner. It employs XML syntax as its underlying data structure and enhances XML expressive power using Declarative Description theory. Computation with XDD is performed by means of Equivalent Transformation (ET), a computational paradigm based on semantic preserving transformations, using the ET Compiler, the ET Interpreter and the XML ET Compiler.

We would like to stress that all the above languages have either a textual syntax or an XML syntax but not both (parameter 8 stated in the previous section).

The RDF Database Access Protocol [HP02] proposes a set of primitives for communicating with a database of RDF data, which include general database management (create, drop, use) addition and deletion of triples, quantification (foreach-where-do, introduce)

[29] See <http://139.91.183.30:9090/RDF/>.
[30] See <http://www.db.ucsd.edu/People/michalis/notes/O2/OQLTutorial.htm>.
[31] See <http://sesame.aidministrator.nl/>.
[32] See <http://www.dfki.uni-kl.de/ruleml/>.
[33] See <http://kr.cs.ait.ac.th/XDD/>.

query response (returns a variable or a triple) and transactional grouping (i.e. operations in the group can be rolled back). Triples may contain variables but conditions are not very sophisticated (Boolean disjunction is included) and they should be user-defined. The concrete syntax can be either textual or XML-like as needed.

The RDFPath[34] proposal aims at making an XPath-like query system for the RDF data model. The purpose of RDFPath is to provide for a general technique to specify paths between two arbitrary nodes of an RDF graph (i.e. to localize information in an RDF graph). An RDFPath expression is build upon three constructions: a *primary selection* specifies an initial set of nodes of a given RDF graph, a composition of *location steps*, each specifying a set of nodes which can be reached by "one step" from a given context object, defines a location path from the initial set of nodes, a *filter* selects a subset of a given set of objects. Primary selections include *resource* and *literal*, location steps include *child*, *parent*, *element*, *container*, *self* and *property* but others could be added, filters seem to be based on Boolean equality of strings but other operators could surely be added. A specific operator is provided to handle transitive closures (as defined in RDF Schema). The proposal seems very primitive at the moment.

TRIPLE[35] is an RDF query, inference, and transformation language for the Semantic Web which allows to define the semantics of languages on top of RDF (like RDF Schema, Topic Maps, UML, etc.) using suitable rules. Access to external programs, like description logics classifiers, is provided in case the description by rules is not easily possible (i.e. DAML+OIL). As a result, TRIPLE allows RDF reasoning and transformation under several semantics allowing to access multiple data source in one application. The query engine is implemented in Java and the language has both a textual and an XML syntax but, at the moment, no complete language description exists (as stated in the 2002-03-14 release documentation).

Finally we would like to mention Metalog[36] which is proposed by M. Marchiori and J. Saarela (both working at W3C). The documentation we could retrieve about Metalog shows that the language is still under development (no complete description of its syntax is given by the authors yet).

[34] See <http://logicerror.com/RDFPath>.

[35] See <http://triple.semanticweb.org/>.

[36] See <http://www.w3.org/RDF/Metalog/paper981007.html>.

Chapter 2

MathQL level 1

In this chapter we discuss the MathQL proposal, briefly announced in [GS03], which concerns a language developed in the context of the HELM project (see Chapter 3) to query RDF metadata about mathematical resources.

2.1 Introduction

2.1.1 Design goals and main features

The MathQL proposal rises within the HELM project with the final aim of providing a set of query languages for digital libraries of formalized mathematical resources, capable of expressing content-aware requests.

This proposal has several domains of application and may be useful for database or on-line libraries reviewers, for proof assistants or proof-checking systems, and also for learning environments because these applications require features for classifying, searching and browsing mathematical information in a semantically meaningful way.

As the most natural way to handle content information about a resource is by means of metadata, our first task is providing a query language that we call MathQL level 1 (or MathQL-1 for short), suitable for a metadata framework. Other languages to be defined in the context of the MathQL proposal may be suitable for queries about the semantic structure of mathematical data: this includes content-based pattern-matching (MathQL-2) and possibly other forms of formal matching involving for instance isomorphism, unification and δ-expansion[1] (MathQL-3).

[1] by δ-expansion we mean the expansion of definitions.

In this perspective the role of a query on metadata can be that of producing a filtered knowledge base containing relevant information for subsequent queries of other kind.

The present work concerns just MathQL-1 which is designed to achieve the following:

1. Exploitation of RDF technology to manage metadata and **compliance with the main requirements** for an RDF query language stated in Section 1.4.1:
   - MathQL-1 provides facilities for hierarchical constraints based on RDF Schema and for traversal of compound values of properties.
   - MathQL-1 provides a full set of Boolean operators to compose query constraints and facilities for selecting URI's or literals by means of regular expressions.
   - MathQL-1 allows to customize the query results specifying what part of a solution should be preserved or discarded.
   - MathQL-1 has a well-conceived semantics that will be presented in this chapter (see Section 2.2), which is defined in term of an abstract metadata model, imposes that queries return exhaustive solutions and includes a "select-from-where"-like construction.
   - MathQL-1 supports a machine-processable XML syntax as well as a human-readable textual syntax to achieve the best usability.
2. Careful **treatment of query results** that should be considered as important as queries themselves. In particular:
   - MathQL-1 query results have a 4-dimensional geometry whereas other languages assume that query results are returned in 1-dimensional structures (i.e. lists of resources) or 2-dimensional structures (i.e. relational database tables). This allows to get better outcomes from queries returning structured results.
   - Besides the syntax for queries, MathQL-1 provides a syntax for query results with its own rigorously defined semantics. This is because, in the context of a distributed setting where query engines are implemented as stand-alone components, non only queries but also query results must travel inside the system and thus need to be encoded in clearly defined format.
3. Exploitation of **constructions borrowed from programming languages** to allow sophisticated queries that need computation over the queried data. In particular:

- MathQL-1 supports variables for storing intermediate query results, provides iterators over these results, has a conditional operator and includes logging facilities for debugging purposes.

As we see, MathQL-1 aims at making up for two limitations (the insufficient compliance with the most requested features and the poor attention paid to query result management) that seem to characterize several implementations and proposals of current RDF query languages (see section 1.4.2).

2.1.2 Architectural issues

This section outlines the main guidelines concerning MathQL-1 general architecture introducing its abstract metadata model and other minor issues.

Attributed values.

The data representation model used by MathQL-1 relies on the notion of *attributed value* (a.v. for short) consists of a *subject string*[2] and an optional set of *named attributes* each holding a *multiple string value* (m.s.v. for short). Attribute names are made of a (possibly empty) list of string components, so they can be hierarchically structured. Moreover the set of attributes is partitioned into *groups* (i.e. subsets) to improve the a.v. structure.

This structure is the building block of query results and MathQL-1 uses it to represent many kinds of information as for instance:

1. A pool of RDF triples having a common subject r, which in general is a URI reference [URI][3], is encoded in a single a.v. placing r in the subject string. the predicates of the triples are encoded as attribute names and their objects are placed in the attributes' values. These values are structured as multiple strings with the aim of holding the objects of repeated predicates. Moreover structured attribute names can encode various components of structured properties preserving their semantics.

   Figure 2.1 shows how a set of triples can be coded in an a.v.. Note that the word *attr* separates the subject string from its attributes, braces enclose an attribute group in which attributes are separated by semicolons, and an equal sign separates an attribute name from its values (see Subsection 2.2.3 for the complete a.v. syntax).

[2] When we say *string*, we mean a finite sequence of characters.

[3] A *URI reference* is a URI with an optional fragment identifier.

```
The RDF triples:
("http://www.w3.org/2002/01/rdf-databases/protocol", "dc:creator", "Sandro Hawke")
("http://www.w3.org/2002/01/rdf-databases/protocol", "dc:creator", "Eric Prud'hommeaux")
("http://www.w3.org/2002/01/rdf-databases/protocol", "dc:date", "2002-01-08")

The corresponding attributed value:
"http://www.w3.org/2002/01/rdf-databases/protocol" attr
          {"dc:creator" = "Sandro Hawke", "Eric Prud'hommeaux"; "dc:date" = "2002-01-08"}
```

Figure 2.1: The representation of a pool of RDF triples

In this setting the grouping feature can be used to separate semantically different classes of properties associated to a resource (as for instance Dublin Core metadata, Euler metadata and user-defined metadata).

Note that the use of a.v.'s to build query results allows MathQL-1 queries to return sets of RDF triples instead of mere sets of resources, in the spirit of what is currently done by other RDF query languages (see Section 1.4.2).

2. The value of a property is encoded in a single a.v. distinguishing three situations:
   - If the property is unstructured, its value is placed in the a.v. subject string and no attributes are defined.
   - If the property is structured and its value has a main component[4], the content of this component is placed in the a.v. subject string and the other components are stored in the a.v. attributes as in the other case.
   - If the property is structured and its value does not have a main component, the a.v. subject string is empty and the components are stored in the attributes.

   Figure 2.2 (first example) shows three possible ways of representing in a.v.'s an instance of a structured property *id* whose value has two fields (i.e. properties) *major* and *minor*. In this instance, *major* is set to "1" and *minor* is set to "2". The representations depend on which component of *id* is chosen as the main component (none, *major* or *minor* respectively). Several structured property values sharing a common main component can be encodes in a single a.v. exploiting the grouping facility: in this case the attributes of every instance are enclosed in separate groups.

[4]Which is set by the *rdf:value* property or defined by a specific application.

```
First example, one instance:
"" attr {"major" = "1"; "minor" = "2"};  no main component
"1" attr {"minor" = "2"};                main component is "major"
"2" attr {"major" = "1"}                 main component is "minor"

Second example: two separate instances:
"" attr {"major" = "1"; "minor" = "2"}, {"major" = "1"; "minor" = "7"}; no main component
"1" attr {"minor" = "2"}, {"minor" = "7"}                        main component is "major"

Third example: two mixed instances:
"" attr {"major" = "3", "6"; "minor" = "4", "9"} no main component
```

Figure 2.2: The representation of the structured value of a property

Figure 2.2 (second example) shows the representations of two instances of *id*: the previous one and a new one for which *major* is "1" and *minor* is "7".

Note that if the attributes of the two groups are encoded in a single group, the notion of which components belong to the same property value can not be recovered in the general case because the values of an attribute form a set and thus are unordered. As an example think of two instances of *id* encoded as in Figure 2.2 (third example).

3. Attributed values can be used to store any auxiliary information needed during query execution. In particular, MathQL-1 provides variables for a.v.'s which, in its textual syntax, are identifiers[5] preceded by the @ sign, as in `@variable`, and that are introduced by the `for` and `select` constructions to be explained below.

The basic operation between a.v.'s is called *addition* and builds a single a.v. starting from two a.v.'s with the same subject string. The subject of the result is always set to the common subject of the operands, but there are two ways to compose the attribute groups:

- With the *set-theoretic* addition, the set of attribute groups in the resulting a.v. is the set-theoretic union of the sets of attribute groups in the operands.
- With the *distributive* addition, the set of attribute groups in the resulting a.v. is the "Cartesian product" of the sets of attribute groups in the two operands. In this context, an element of the "Cartesian product" is not a pair of groups but it is the set-theoretic union of these groups where the m.s.v.'s of homonymous attributes are clustered together using set-theoretic unions.

[5]To be understood as in programming languages.

```
attributed values used as operands for the addition:
"1" attr {"A" = "a"}, {"B" = "b1"}
"1" attr {"A" = "a"}, {"B" = "b2"}

Set-theoretic addition:
"1" attr {"A" = "a"}, {"B" = "b1"}, {"B" = "b2"}

Distributive addition:
"1" attr {"A" = "a"}, {"B" = "b1", "b2"}, {"A" = "a"; "B" = "b2"}, {"B" = "b1"; "A" = "a"}
```

Figure 2.3: The addition of attributed values

Figure 2.3 shows an example of the two kinds of addition.

Query results.

The result of a MathQL-1 query is always a set of a.v.'s whose subject strings are distinct. MathQL-1 defines three operations on a.v. sets, namely:

- The *union* corresponds to the set-theoretic union where the a.v.'s sharing a common subject are packed in single a.v.'s adding them set-theoretically as explained above. This operation plays a central role MathQL-1 architecture and allows to compose the attributes of the operands preserving their group structure.
- The *intersection* contains the set-theoretic union of the a.v.'s whose subject string appears in each argument. In this case the a.v.'s sharing a common subject are packed in single a.v.'s adding them distributively. This strong form of intersection has the double benefit of filtering the common subjects of the given a.v. sets, and of merging their attribute groups in every possible way. This feature enables the possibility of performing additional filtering operations checking the content of the merged groups.
- The *difference* of two a.v. sets contains the a.v.'s of the first argument whose subject string does not appear in the second argument.

Following the previous intuition, an a.v. set can be used to represent a set of RDF triples or a set of property values: this is done by representing each triple or value in a single a.v. and then by combining these a.v.'s with the union operation we just defined.

```
"A" attr {"major" = "1"; "minor" = "2"},
         {"first" = "2002-01-01"; "modified" = "2002-03-01"};
"B" attr {"major" = "1"; "minor" = "7"},
         {"first" = "2002-02-01"; "modified" = "2002-04-01"}
```

	"major"	**"minor"**	**"first"**	**"modified"**
"A"	"1"	"2"	"2002-01-01"	"2002-03-01"
"B"	"1"	"7"	"2002-02-01"	"2002-04-01"

Figure 2.4: A set of attributed values displayed as a table

```
The wanted set of attributed values contains this element:
"A" attr {"major" = "1"; "minor" = "2"}

This query generates the wanted set:
add "1" as "major", "2" as "minor" in subj "A"
```

Figure 2.5: Building a simple set of attributed values

If the a.v.'s of an a.v. set share the same attribute names and grouping structure, this set can be represented as a table in which each row encodes an a.v. and each column is associated to an attribute (except the first one which holds the subject strings). Figure 2.4 shows an a.v. set describing the properties of two resources "A" and "B" giving its table representation, in which the columns corresponding to attributes in the same group are clustered between double-line delimiters[6].

The above example gives a spatial idea of the geometry of an a.v. set (i.e. a query result) which fits in 4 dimensions: namely we can extend independently the set of the subject strings (dimension 1), the attributes in each group (dimension 2), the groups in each a.v. (dimension 3) and the set of strings stored in each attribute value (dimension 4).

The metadata defined in the table of Figure 2.4 will be used in subsequent examples. For this purpose assume that *first* and *modified* are the components of a structured property *date* available for the resources "A" and "B".

MathQL-1 support for a.v. set manipulation includes the following constructions, in which *query* is an a.v. set and *value* is a m.s.v. (the precise syntax is in Subsection 2.2.2):

- `empty`: builds the empty a.v. set.

[6] A table with grouped labelled columns like the one above resembles a set of relational database tables.

- `subj` *value*: builds an a.v. set taking the subject strings from *value*. Each resulting a.v. has an empty set of attributes.
- `add` *optional-flag attribute-groups* `in` *query*: builds an a.v. set adding the specified *attribute-groups* (contained in a virtual a.v.) to each a.v. of *query*. If no *flag* is specified the addition is set-theoretic, whereas with the `distr` flag the addition is distributive. The *groups* can be explicit or they can be red from a variable for a.v.'s.

  Figure 2.5 shows how to build a one-element a.v. set using `subj` and `add`.
- `proj` *optional-attribute-name query*: builds an m.s.v. containing either the subject strings of *query*[7] (if the *attribute-name* is not specified) or the values, searched in each group and composed by set-theoretic union, found in *query* for the specified attribute[8].
- `keep` *optional-flag attribute-name-list* `in` *query*: builds an a.v. set by removing from *query* every attribute whose name is included (or is not, according to the *flag*) in the given *attribute-name-list*. If the *flag* is not present, the *list* specifies the attributes to keep, whereas if the *flag* is `allbut`, the *list* specifies the attributes to remove. Removing unwanted information from an a.v. set is useful in two cases: lowers the complexity of intermediate query results increasing the performance of subsequent operations and cleans the final query results making them easier to manage for the application that submitted the query.
- *query infix-binary-operator query*: builds an a.v. set by composing the two operands according to the *operator* which can be `union`, `intersect` or `diff`. The composition occurs as explained above.
- `let` *query-variable* `be` *query* `in` *query*: stores the result of the first *query* in a variable for a.v. sets before evaluating the second *query*. Variables for a.v. sets are represented by identifiers preceded by the % sign, as in `%variable`.

Ordering of query results.

The data stored in a.v. sets are formally unordered but a MathQL-1 query engine may implement facilities for presenting query results as ordered structures.

[7] This is the content of the first column of *query* viewed as a table.

[8] This is the content of a labelled column of *query* viewed as a table.

Accessing RDF metadata.

Formally MathQL-1 allows to access an RDF graph[9] through an *access relation* which is better understood by explaining the informal semantics of the `property` operator.
This operator builds a *result* a.v. set starting from two mandatory arguments: the *source* m.s.v. and the *head path*. Other optional arguments may be used to change its default behaviour or to request advanced functionalities. The textual syntax of this operator is:
`property` *optional-flags head-path optional-clauses* `of` *optional-flag value*
A path has the structure of an attribute name (i.e. a list of strings) and denotes a (possibly empty) finite sequence of contiguous arcs (describing properties in the RDF graph).

In the simplest case `property` is used to read the values of a (possibly compound) property with an unstructured value and does the following:

- It computes the instances of the given path in the RDF graph available to the query engine, using the resources specified in the source m.s.v. (call them source resources) as start-nodes.
- The computation gives a set of nodes in the RDF graph (i.e. the end-nodes of the instantiated paths) which are the values of the instances of the (possibly compound) property specified by the path and concerning the source resources.
- These values, encoded into a.v.'s as explained above, are composed by means of the MathQL-1 union operation to form the result set.

Figure 2.6 (example 1) shows an instance of this procedure. Note that the result sets of this example have no attributes and that a path is represented by a slash-separated list of strings denoting the path's arcs.[10]

Using the `pattern` flag, `property` can be instructed to regard the contents of the source m.s.v. as POSIX regular expressions rather than as constant strings. In this case `pattern` selects the set of resources matching at least one of the given expressions. See for instance Figure 2.6 (example 2).

If we want to read the value of a structured property we can specify the value's main component in the `main` *optional-clause* (this specification overrides the default setting inferred from the RDF graph through the *rdf:value* property) and the list of the value's

[9] When we say RDF graph, we actually mean both the RDF Model graph and the RDF Schema graph.
[10] If needed, the empty path is represented by a single slash.

```
These examples refer to the resources "A" and "B" of Figure 2.4.

Example 1: reading an unstructured property - simple case:
property "id"/"major" of {"A", "B"}   returns   "1"
property "id"/"minor" of {"A", "B"}   returns   "2"; "7"

Example 2: reading an unstructured property - use of pattern:
property "id"/"minor" of pattern ".*"   returns   "2"; "7"

Example 3: reading a structured property without main component:
property "id" attr "major", "minor" of {"A", "B"}
generates the following attributed values:
"" attr {"major" = "1"; "minor" = "2"}; "" attr {"major" = "1"; "minor" = "7"}
that are composed using MathQL-1 union giving the one-element set:
"" attr {"major" = "1"; "minor" = "2"}, {"major" = "1"; "minor" = "7"}

Example 4: reading a structured property specifying a main component:
property "id" main "major" attr "minor" of {"A", "B"}   gives
"1" attr {"minor" = "2"}, {"minor" = "7"}

Example 5: the renaming mechanism:
property "id" attr "minor" as "new-name" of {"A", "B"}   gives
"" attr {"new-name" = "2"}, {"new-name" = "7"}

Example 6: imposing constraints on property values:
property "date" istrue "first" in "2002-01-01" attr "modified" of {"A", "B"}   and
property "date" istrue "first" match ".*01.*" attr "modified" of {"A", "B"}    give
"" attr {"modified" = "2002-03-01"}
Only the instance of "date" with "first" set to "2002-01-01" is considered.

Example 7: inverse traversal of the head path:
property inverse "date" attr "first" in subj ""   gives
"A" attr {"first" = "2002-01-01"}; "B" attr {"first" = "2002-02-01"}

Example 8: some triples of an access relation:
The triples formalizing the property "date" of the resource "A":
("A", "date", "");
("A", "date"/"first", "2002-01-01"); ("A", "date"/"modified", "2002-03-01")
```

Figure 2.6: The “property” operator

secondary components in the `attr` *optional-clause*. Note that if a secondary component is not listed in the `attr` clause, it will not be read. Also recall that, when the result a.v.'s are formed, the main component is is read in the subject string, whereas the secondary components are encoded using the attributes of a single group. See for instance Figure 2.6 (examples 3 and 4). As a component of a property's value may be a structured property, its specification (appearing in the `main` or `attr` clause) is actually a path in the RDF graph starting from the end-node of the head path.

Note that the name of an attribute, which by default is its defining path in the `attr` clause, can be changed with an optional `as` clause for the user's convenience. See for instance Figure 2.6 (example 5). The alternative could be a simple string but needs to be a path for typing reasons. In any case a string can be seen as a one-element path.

In the default case `property` builds its result considering every component of the RDF Model graph (i.e. every RDF Model) but we can constrain some nodes of the inspected components to have (or not to have) a given value, with the aim of improving the performance of the inspection procedure. The constrained nodes are specified in the `istrue` and `istrue` *optional-clauses* and the constraining values are expressed by `in` or `match` constructions depending on their semantics (constant values or POSIX regular expressions respectively). See for instance Figure 2.6 (example 6). Again a constrained node may be the value of a compound property, therefore its specification (appearing in the `istrue` or `isfalse` clause) is a path in the RDF graph starting from the end-node of the head path.
`property` allows to access the RDF Schema property hierarchy by specifying a flag named `sub` or `super`. If the `sub` flag is present, `property` inspects the instances of the default tree (made by the head path and by the *optional-clauses* paths) and every other tree obtained by substituting an arc p with the arc of a subproperty of p. If the `super` flag is present, super-property arcs are substituted instead.
`property` also allows the inverse traversal of its head path if the `inverse` flag is specified. In this case the operator works as follows:

- It instantiates the head path using the values whose main component is specified in the source m.s.v. set as end-nodes.
- It encodes the resources corresponding to the instances of the start-nodes into a.v.'s assigning the attributes obtained instantiating the attribute paths[11] and composes

[11] The path in *optional-clauses* are never traversed backward.

these a.v.'s using the MathQL-1 union operation to build the result set.

See for instance Figure 2.6 (example 7).

Now we can present *access relations* which are the formal tools used by MathQL-1 semantics to access the RDF graph. An access relation is a set of triples (r_1, p, r_2) where r_1 and r_2 are strings, p is a path (encoded as a list of strings). Each triple is a sort of "extended RDF triple" in the sense that r_1 is is a resource for which metadata is provided, p is a path in the RDF graph and r_2 is the main value of the end-node of the instance of p starting from r_1 (this includes the instances of sub- and super-arcs of p if necessary). See for instance Figure 2.6 (example 8).

MathQL-1 does not provide for any built-in access relation so any query engine can freely define the access relations that are appropriate with respect to the metadata it can access. In particular, Section 4.1.2 describes the access relations implemented by the HELM query engine.

It is worth remarking, as it was already stressed in [GS03], that the concept of access relation corresponds to the abstract concept of property in the basic RDF data model which draws on well established principles from various data representation communities. In this sense an RDF property can be thought of either as an attribute of a resource (traditional attribute-value pairs model), or as a relation between a resource and a value (entity-relationship model). This observation leads us to conclude that MathQL-1 is sound and complete with respect to querying an abstract RDF metadata model.

Finally note that access relations are close to RDF entity-relationship model, but they do not work if we allow paths with an arbitrary number of loops (i.e. with an arbitrary length) because this would lead to creating infinite sets of triples. If we want to handle this case, we need to turn these relations into multivalued functions.

Multiple string values, Boolean values and numbers.

Multiple string values (i.e. sets of strings), are the most important data structures after a.v.'s and a.v. sets. MathQL-1 uses these structures to encode string data, Boolean values and natural numbers. More precisely:

- When interpreting a m.s.v. as a Boolean value, the empty set is regarded as *false*

while any inhabited set is regarded as *true*.[12] A specific inhabited m.s.v. (the one-element set containing the empty string) is used as the default encoding of *true*.

- Numbers are encoded with m.s.v.'s containing the string of their decimal expansion.

MathQL-1 support for m.s.v. manipulation includes the following constructions:

- `{`*value*`, ···, `*value*`}`: builds the set-theoretic union of the specified m.s.v.'s.
- `count` *value*: builds an m.s.v. containing the size of the *value*.
- `false`: builds the m.s.v. representing the *false* Boolean value (the empty m.s.v.).
- `true`: builds the default m.s.v. representing a *true* Boolean value (*default-true*).
- *value infix-test-operator value*: builds *false* or *default-true* according to the specified test. The *test-operator* includes: `sub` (set-theoretic subset relation), `eq` (set-theoretic quality), `meet` (inhabitance of the set-theoretic intersection), `le` (numeric less-or-equal-than), `lt` (numeric less-than).[13]
- `not` *value*: returns *false* if the *value* is *true*, and *default-true* otherwise.
- *value* `and` *value*: returns the second *value* if the first *value* is *true*, and *false* otherwise.
- *value* `or` *value*: returns the first *value* if it is *true*, and the second *value* otherwise.
- *value* `xor` *value*: returns *false* if both values are *true* or *false*, and the *true value* otherwise.
- `let` *value-variable* `be` *value* `in` *query*: stores the *value* in a variable for values before evaluating the *query*. Variables for m.s.v.'s are represented by identifiers preceded by the $ sign, as in `$variable`.

The `ex` and "dot" operators are used to read the attributes of av's stored in variables for av's. They will be discussed in Subsection 2.2.2.

Conditional queries and iterators.

The following constructions allow sophisticated queries like the one in Subsection 4.2.3:

- `if` *value* `then` *query* `else` *query*: executes one of the specified *queries* interpreting the *value* with the Boolean semantics described above.

[12] This choice was inspired by the C-style encoding of Boolean values on top of integer numbers.

[13] `le` and `lt` build *false* if their operands are invalid numbers.

- `for` *av-variable* `in` *query* `sup` *query*: iterates the evaluation of the second *query* setting the *av-variable* to each a.v. in the first *query* and builds the MathQL-1 union of the obtained results.
- `for` *av-variable* `in` *query* `inf` *query*: like the former but MathQL-1 intersection is used instead of MathQL-1 union.
- `select` *av-variable* `from` *query* `where` *value*: This is the well-known "select-from-where" construction suggested by the RDF community (see Subsection 1.4.1).

  Its semantics is the one of:

  `for` *av-variable* `in` *query* `sup if` *value* `then` *av-variable* `else empty`.

Logging facilities.

MathQL-1 provides for three logging facilities which where introduced mainly for debugging purposes. In particular the language includes these constructions:

- `log` *optional-flags query*: logs and returns the *query*. Normally the result of the *query* is logged, but with the `source` flag, the *query* itself is logged. By default, logging occurs in textual syntax but the `xml` flag switches to XML-logging mode.
- `stat` *query*: returns the *query* logging the size of its result and the execution time.
- `stat` *value*: returns the *value* logging its size and its evaluation time.

Constant strings.

As we saw in the above examples, MathQL-1 represents constant strings surrounding them with double quotes. The general attitude of the language towards constant strings is to consider them as data to be processed verbatim, in this sense MathQL-1 semantics provides only for the escaping and unescaping of special characters occurring in these strings. MathQL-1 character escaping syntax aims at complying with W3C character model for the World Wide Web [W3Ca] which recommends a support for standard [Unicode] characters (U+0000 to U+FFFF) and escape sequences with start/end delimiters. In particular MathQL-1 escape delimiters (backslash and caret) are chosen among the *unwise* characters for URI references (see [URI]) because URI references are the natural content of constant strings and these characters should not be so frequent in them.

2.2 Operational semantics

This section describes the current state of MathQL-1 syntax and semantics, which is still unstable and is maintained by us in collaboration with I. Schena. Section 2.3 outlines the differences between this version of the language and the older versions presented in [GS03, Nat02, Lor02].

We present MathQL-1 semantics in a natural operational style [Lan98, Win93] and we use a simple type system that includes basic types such as strings and Booleans, and some type constructors such as product and exponentiation. $y : Y$ will denote a typing judgement. This semantics is not meant as a formal system *per se*, but should serve as a reference for implementors. In this sense an interesting property that could be proved formally is that the evaluation of a query is deterministic and always terminating.[14] However the present work does not contain such a formal proof.

2.2.1 Mathematical background

`String` denotes the type of strings and its elements are the finite sequences of [Unicode] characters. Grammatical productions, represented as strings in angle brackets, denote the subtype of `String` containing the produced sequences of characters.

The syntax of grammatical productions resembles BNF and POSIX notation:

- `::=` defines a grammatical production by means of a regular expression.

Regular expressions are made of the following elements (here `...` is a placeholder):

- `‘...‘` represents any character in a character set;
- `‘^ ...‘` represents any character (U+0020 to U+007E) not in a character set;
- `"..."` represents a string to be matched verbatim;
- `<...>` represents a regular expression defined by a grammatical production;
- `... ...` represents a conjunctive regular expression;
- `... | ...` represents a disjunctive regular expression;
- `[ ... ]?` represents an optional regular expression;
- `[ ... ]+` represents a regular expression to be repeated one or more times;

[14] As a consequence, the language is not Turing-complete.

- `[ ... ]*` represents a regular expression to be repeated zero or more times;
- `[ ... ]` represents a grouped regular expression.

`Num` denotes the type of numbers and is defined as the subtype of `String` given by the regular expression: `'0 - 9' [ '0 - 9' ]*`. In this type, numbers are represented by their decimal expansion.

`SetOf` Y denotes the type of finite sets (i.e. unordered finite sequences without repetitions) over Y. `ListOf` Y denotes the type of lists (i.e. ordered finite sequences) over Y. We will use the notation $[y_1, \cdots, y_m]$ for the list whose elements are $y_1, \cdots, y_m$.
`Boole` denotes the type of Boolean values and is defined as $\{\emptyset, \{""\}\}$: `SetOf SetOf String` where $\emptyset$: `SetOf String` is the *false* value (denoted by `F`) and the one-element set $\{""\}$: `SetOf String` is the *true* value (denoted by `T`).

$Y * Z$ denotes the product of the types Y and Z whose elements are the ordered pairs (y, z) such that $y : Y$ and $z : Z$. The notation is also extended to a ternary product.

$Y \to Z$ denotes the type of functions from Y to Z and $f\ y$ denotes the application of $f : Y \to Z$ to $y : Y$. Relations over types, such as equality, are seen as functions to `Boole`.

With the above constructors we can give a formal meaning to most of the standard notation. For instance we will use the following:

- $\emptyset$: (`SetOf` Y)
- $\exists$: ((`SetOf` Y) $\to$ `Boole`) $\to$ `Boole`
- $\forall$: ((`SetOf` Y) $\to$ `Boole`) $\to$ `Boole`
- $\in$: $Y \to$ (`SetOf` Y) $\to$ `Boole` (infix)
- $\subseteq$: (`SetOf` Y) $\to$ (`SetOf` Y) $\to$ `Boole` (infix)
- $\not\subset$: (`SetOf` Y) $\to$ (`SetOf` Y) $\to$ `Boole` (infix)
- $\cap$: (`SetOf` Y) $\to$ (`SetOf` Y) $\to$ (`SetOf` Y) (infix)
- $\cup$: (`SetOf` Y) $\to$ (`SetOf` Y) $\to$ (`SetOf` Y) (infix)
- $\sqcup$: (`SetOf` Y) $\to$ (`SetOf` Y) $\to$ (`SetOf` Y) (the disjoint union, infix)
- $\leq$: `Num` $\to$ `Num` $\to$ `Boole` (infix)
- $<$: `Num` $\to$ `Num` $\to$ `Boole` (infix)
- $\#$: (`SetOf` Y) $\to$ `Num` (the size operator)

- $@ : (\texttt{ListOf}\ Y) \to (\texttt{ListOf}\ Y) \to (\texttt{ListOf}\ Y)$ (the concatenation, infix)
- $\neg : \texttt{Boole} \to \texttt{Boole}$

Note that $\forall$ and $\exists$ are always decidable because the sets are finite by definition.
$U \between W$ means $(\exists u \in U)\ u \in W$ and expresses the fact that $U \cap W$ is inhabited as a primitive notion, i.e. without mentioning intersection and equality as for $U \cap W \neq \emptyset$, which is equivalent but may be implemented less efficiently in real cases[15].

$U \between W$ is a natural companion of $U \subseteq W$ being its logical dual (recall that $U \subseteq W$ means $(\forall u \in U)\ u \in W$) and is already being used successfully in the context of a constructive (i.e. intuitionistic and predicative) approach to point-free topology [Sam00].

Sets of couples play a central role in our formalization and in particular we will use:

- $\texttt{Fst} : (Y \times Z) \to Y$ such that $\texttt{Fst}\ (y, z) = y$.
- $\texttt{Snd} : (Y \times Z) \to Z$ such that $\texttt{Snd}\ (y, z) = z$.
- With the same notation, if W contains just one couple whose first component is y, then $W(y)$ is the second component of that couple. In the other cases $W(y)$ is not defined. This operator has type $(\texttt{SetOf}\ (Y \times Z)) \to Y \to Z$.
- Moreover $W[y \leftarrow z]$ is the set obtained from W removing every couple whose first component is y and adding the couple (y, z). The type of this operator is $(\texttt{SetOf}\ (Y \times Z)) \to Y \to Z \to (\texttt{SetOf}\ (Y \times Z))$.
- Also $U + W$ is the union of two sets of couples in the following sense:

$$\begin{array}{rcl} U + \emptyset & \text{rewrites to} & U \\ U + (W \sqcup \{(y, z)\}) & \text{rewrites to} & U[y \leftarrow z] + W \end{array}$$

The last three operators are used to read, write and join association sets, which are sets of couples such that the first components of two different elements are always different. These sets will be exploited to formalize the memories appearing in evaluation contexts.

Now we are able to type the main objects needed in the formalization:

- A path s is a list of strings therefore its type is $T_{0a} = \texttt{ListOf String}$.
- A multiple string value V is an object of type $T_{0b} = \texttt{SetOf String}$.
- A attribute group G is an association set connecting the attribute names to their values, therefore its type is $T_1 = \texttt{SetOf}\ (T_{0a} \times T_{0b})$.

[15] As for the Boolean condition $\phi \vee \psi$ which may have a more efficient implementation than $\neg(\neg\phi \wedge \neg\psi)$.

```
<dec>         ::= '0 - 9'
<num>         ::= <dec> [ <dec> ]*
<hex>         ::= <dec> | 'A - F' | 'a - f'
<escaped>     ::= "u" <hex> <hex> <hex> <hex> | '"' | "\" | "^"
<string>      ::= '"' [ "\" <escaped> "^" | '^ "\^' ]* '"'
<string_list> ::= <string> [ "," <string> ]*
<path>        ::= [ "/" ]? <string> [ "/" <string> ]* | "/"
```

Figure 2.7: Textual syntax of numbers, strings and paths

Escape sequence	Unicode character	Text
\u....^	U+....	
\"^	U+0022	"
\\^	U+005C	\
\^^	U+005E	^

Figure 2.8: Textual syntax of escaped characters

- A subject string r is an object of type `String`.
- A set A of attribute groups is an object of type $T_2 =$ `SetOf` T_1.
- An a.v. is a subject string with its attribute groups, so its type is $T_3 =$ `String` $\times T_2$.
- A set S of a.v.'s is an object of type $T_4 =$ `SetOf` T_3.
- A triple of an attributed relation is of type $T_5 =$ `String` $\times$ `String` $\times (T_{0a} \rightarrow$ `String`$)$.

We will also need some primitive functions that mostly retrieve the information that an implemented query engine obtains reading its underlying database. These functions will be explained in the next section when needed.

2.2.2 Textual syntax and semantics of queries

MathQL-1 expressions denoting queries fall into three categories.

- Expressions denoting an a.v. set belong to the grammatical production `<query>` and their semantics is given by the infix evaluating relation $\Downarrow_q$.
- Expressions denoting a multiple string value belong to the grammatical production `<value>` and their semantics is given by the infix evaluating relation $\Downarrow_v$.

Expressions can contain quoted constant strings with the syntax of Figure 2.7.

```
<alpha>  ::= [ 'A - Z' | 'a - z' | '_' ]+
<id>     ::= <alpha> [ <alpha> | <dec> ]*
<avar>   ::= "@" <id>
<qvar>   ::= "%" <id>
<vvar>   ::= "$" <id>
```

Figure 2.9: Textual syntax of variables

When these strings are unquoted, the surrounding double quotes are deleted and each escaped sequence is translated according to the table in Figure 2.8 (where .... is a 4-digit placeholder). This operation is formally performed by the function `Unquote` of type `String` $\rightarrow$ `String`. Moreover `Name` : `<path>` $\rightarrow T_{0a}$ is a helper function that converts a linearized path in its structured representation.
Formally `Name` $(q_1$ / $\cdots$ / $q_m)$ rewrites to [`Unquote` $q_1, \cdots,$ `Unquote` q_m].
Note that in the present version of the language, the slash at the beginning of a path is semantically irrelevant, so for instance `/"my"/"name"` is equivalent to `"my"/"name"`.

Query expressions can contain variables for a.v.'s (avar), variables for a.v. sets, i.e. for query results (qvar) and variables for multiple strings values (vvar).

Query expressions are evaluated in a context $\Gamma = (\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v)$ which is a quadruple of association sets that connect qvar's to a.v. sets, avar's to a.v.'s, avar's to attribute groups and vvar's to multiple string values. Therefore the type K of the context Γ is:

$$\texttt{SetOf}\ (\texttt{<qvar>} \times T_4) \times \texttt{SetOf}\ (\texttt{<avar>} \times T_3)\ \times \texttt{SetOf}\ (\texttt{<avar>} \times T_1) \times \texttt{SetOf}\ (\texttt{<vvar>} \times T_{0b})$$

and the three evaluating relations are of the following types:

$$\Downarrow_q : (K \times \texttt{<query>}) \rightarrow T_4 \rightarrow \texttt{Boole},$$
$$\Downarrow_v : (K \times \texttt{<value>}) \rightarrow T_{0b} \rightarrow \texttt{Boole}.$$

The context components Γ_q, Γ_a and Γ_v are used to store the contents of variables, while Γ_g is used by the `ex` Boolean operator to be presented below.

Expressions denoting an a.v. set

These expressions represent queries or sub-queries and their syntax is described in Figure 2.10 (the start symbol is `<query>`).

The first bunch of `<query>` operators gives the helper functionalities for context manipulation, syntactic grouping and logging. In particular:

- The `let` operators:

```
<def>       ::= <path> "as" <value>
<groups>    ::= <avar> | <def> [ "," <def> ]*
<path_list> ::= <path> [ "," <path> ]*
<qualifier> ::= [ "inverse" ]? [ "sub" | "super" ]? <path>
<main>      ::= [ "main" <path> ]?
<cons>      ::= <path> [ "in" | "match" ] <value>
<istrue>    ::= [ "istrue" <cons> [ "," <cons> ]* ]?
<isfalse>   ::= [ "isfalse" <cons> [ "," <cons> ]* ]?
<exp>       ::= <path> [ "as" <path> ]?
<sec>       ::= [ "attr" <exp> [ "," <exp> ]* ]?
<opt_args>  ::= <main> <istrue> <isfalse> <sec>
<source>    ::= [ "pattern" ]? <value>
<query>     ::= "let" <qvar> "be" <query> "in" <query>
            |   "let" <vvar> "be" <value> "in" <query>
            |   <qvar> | <avar> | "(" <query> ")"
            |   "log" [ "xml" ]? [ "source" ]? <query>
            |   "stat" <query>
            |   "empty" | "subj" <value>
            |   <query> [ "intersect" | "union" | "diff" ] <query>
            |   "add" [ "distr" ]? <groups> "in" <query>
            |   "keep" [ "allbut" ]? [ <path_list> ]? "in" <query>
            |   "if" <value> "then" <query> "else" <query>
            |   "for" <avar> "in" <query> [ "sup" | "inf" ] <query>
            |   "select" <avar> "from" <query> "where" <value>
            |   "property" <qualifier> <opt_args> "of" <source>
<value>     ::= <string> | <vvar> | "(" <value> ")" | "stat" <value>
            |   "{" [ <value> [ "," <value> ]* ]? "}" | "count" <value>
            |   "proj" [ <path> ]? <query>
            |   <value> [ "sub" | "meet" | "eq" | "le" | "lt" ] <value>
            |   "false" | "true" | "not" <value>
            |   <value> [ "and" | "or" | "xor" ] <value>
            |   "ex" <value> | <avar> "." <path>

Precedence classes (listed from low to high using brackets for grouping):
(let, add, keep, for, if,  log, stat), diff, union, intersect, (select, ex),
(or, xor), and, not, (sub, meet, eq, le, lt), (subj, property, proj, count).
Left-associative operators: intersect, union, diff, and, or, xor.
```

Figure 2.10: Textual syntax of queries

$$\frac{i : \texttt{<qvar>} \quad ((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), x_1) \Downarrow_q S_1 \quad ((\Gamma_q[i \leftarrow S_1], \Gamma_a, \Gamma_g, \Gamma_v), x_2) \Downarrow_q S_2}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), \text{let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_q S_2}$$

$$\frac{i : \texttt{<vvar>} \quad ((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), x_1) \Downarrow_v V \quad ((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v[i \leftarrow V]), x_2) \Downarrow_q S}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), \text{let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_q S}$$

- The operators for reading variables and for syntactic grouping:

$$\frac{i : \texttt{<qvar>}}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), i) \Downarrow_q \Gamma_q(i)} \qquad \frac{i : \texttt{<avar>}}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), i) \Downarrow_q \{\Gamma_a(i)\}} \qquad \frac{(\Gamma, x) \Downarrow_q S}{(\Gamma, (x)) \Downarrow_q S}$$

  $\Gamma_q(i)$ and $\{\Gamma_a(i)\}$ mean $\emptyset$ if i is not defined.

- The logging operators (`log` and `stat`):

$$\frac{h_1 \in \texttt{["xml"]?} \quad h_2 \in \texttt{["source"]?} \quad (\Gamma, x) \Downarrow_q S}{(\Gamma, \log\ h_1\ h_2\ x) \Downarrow_q \texttt{Log}\ h_1\ h_2\ x\ S} \qquad \frac{x : \texttt{<query>}}{(\Gamma, \text{stat } x) \Downarrow_q \texttt{QStat}\ \Gamma\ x}$$

$$\frac{}{\texttt{Log}\ h_1\ h_2\ x\ S \text{ rewrites to } S} \qquad \frac{(\Gamma, x) \Downarrow_q S}{\texttt{QStat}\ \Gamma\ x \text{ rewrites to } S}$$

The second bunch of `<query>` operators gives the functionalities for a.v. set manipulation including set-theoretic operations. In particular:

- The `empty` constant and the `subj` (subject) operator:

$$\frac{}{(\Gamma, \text{empty}) \Downarrow_q \emptyset} \qquad \frac{(\Gamma, y) \Downarrow_v V}{(\Gamma, \text{subj } y) \Downarrow_q \{(v, \emptyset) \mid v \in V\}}$$

  `subj` makes a cast (which we are planning to hide) between the types T_{0b} and T_4.

- The semantics of `union`, `intersect` and `diff` is defined by means of three helper functions $\oplus$, $\otimes$ and $\ominus$, each having two rewrite rules.

$$\frac{(\Gamma, x_1) \Downarrow_q S_1 \quad (\Gamma, x_2) \Downarrow_q S_2}{(\Gamma, x_1 \text{ union } x_2) \Downarrow_q S_1 \oplus S_2} \qquad \frac{(\Gamma, x_1) \Downarrow_q S_1 \quad (\Gamma, x_2) \Downarrow_q S_2}{(\Gamma, x_1 \text{ intersect } x_2) \Downarrow_q S_1 \otimes S_2} \qquad \frac{(\Gamma, x_1) \Downarrow_q S_1 \quad (\Gamma, x_2) \Downarrow_q S_2}{(\Gamma, x_1 \text{ diff } x_2) \Downarrow_q S_1 \ominus S_2}$$

1a	$(S_1 \sqcup \{(r, A_1)\}) \oplus (S_2 \sqcup \{(r, A_2)\})$	rewrites to	$S_1 \oplus S_2 \oplus \{(r, A_1 \cup A_2)\}$
1b	$S_1 \oplus S_2$	rewrites to	$S_1 \cup S_2$
2a	$(S_1 \sqcup \{(r, A_1)\}) \otimes (S_2 \sqcup \{(r, A_2)\})$	rewrites to	$(S_1 \otimes S_2) \cup \{(r, A_1 \odot A_2)\}$
2b	$S_1 \otimes S_2$	rewrites to	$\emptyset$
3a	$(S_1 \sqcup \{(r, A_1)\}) \ominus (S_2 \sqcup \{(r, A_2)\})$	rewrites to	$S_1 \ominus S_2$
3b	$S_1 \ominus S_2$	rewrites to	S_1

  Rules 1a, 2a, 3a take precedence over rules 1b, 2b, 3b respectively, $\oplus$ is defined associative and $A_1 \odot A_2 = \{G_1 \oplus G_2 \mid G_1 \in A_1, G_2 \in A_2\}$.

- The semantics of the `add` operator is defined in terms of the helper function `Add`:

$$\frac{h \in \texttt{["distr"]?} \quad (\Gamma, y_1) \Downarrow_v V_1 \quad \cdots \quad (\Gamma, y_n) \Downarrow_v V_n \quad p_1 : \texttt{<path>} \quad \cdots \quad p_n : \texttt{<path>} \quad (\Gamma, x) \Downarrow_q S}{(\Gamma, \text{add } h\ y_1 \text{ as } p_1, \cdots, y_n \text{ as } p_n \text{ in } x) \Downarrow_q \texttt{Add}\ h\ \{\{(\texttt{Name}\ p_1, V_1)\} \oplus \cdots \oplus \{(\texttt{Name}\ p_n, V_n)\}\}\ S}$$

$$\frac{h \in \texttt{["distr"]?} \quad i : \texttt{<avar>} \quad ((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), x) \Downarrow_q S}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), \text{add } h \ i \text{ in } x) \Downarrow_q \texttt{Add } h \ (\texttt{Snd } \Gamma_a(i)) \ S}$$

1	$\texttt{Add } h \ A_1 \ \emptyset$	rewrites to	$\emptyset$	
2	$\texttt{Add } h \ A_1 \ (\{(r, A_2)\} \sqcup S)$	rewrites to	$\{(r, A_1 \square_h A_2)\} \cup (\texttt{Add } h \ A_1 \ S)$	

Where $\square_{\text{""}} = \cup$ and $\square_{\text{"}\texttt{distr}\text{"}} = \odot$.

- The semantics of the `keep` operator is expressed by the following rules, where W is $\{\texttt{Name } p_1, \cdots, \texttt{Name } p_m\}$. Moreover `Keep` and `Keep'` are two helper functions.

$$\frac{h \in \texttt{["allbut]?} \quad p_1 : \texttt{<path>} \quad \cdots \quad p_k : \texttt{<path>} \quad (\Gamma, x) \Downarrow_q S}{(\Gamma, \text{keep } h \ p_1, \cdots, p_m \text{ in } x) \Downarrow_q \{(r, \bigcup\{\texttt{Keep } h \ W \ G \mid G \in A\}) \mid (r, A) \in S\}}$$

$$\frac{\texttt{Keep'} \ h \ W \ G \text{ rewrites to } \emptyset}{\texttt{Keep} \ h \ W \ G \text{ rewrites to } \emptyset} 1 \qquad \frac{\texttt{Keep'} \ h \ W \ G \text{ rewrites to } G'}{\texttt{Keep} \ h \ W \ G \text{ rewrites to } \{G'\}} 2$$

3	$\texttt{Keep'} \ \texttt{""} \ W \ ((s, V) \sqcup G) \ W$	rewrites to	$\texttt{Keep'} \ \texttt{""} \ W \ G$	if $s \notin W$
4	$\texttt{Keep'} \ \texttt{"allbut"} \ W \ ((s, V) \sqcup G) \ W$	rewrites to	$\texttt{Keep'} \ \texttt{"allbut"} \ W \ G$	if $s \in W$
5	$\texttt{Keep'} \ h \ W \ G$	rewrites to	G	

Here smaller rule numbers indicate higher precedence for the respective rules.

The third bunch of `<query>` operators gives the functionalities for conditional queries and iteration on a.v. sets. In particular:

- Concerning the `if` operator, rule 1 has higher precedence than rule 2:

$$\frac{(\Gamma, y) \Downarrow_v \texttt{F} \quad (\Gamma, x_2) \Downarrow_q S_2}{(\Gamma, \text{if } y \text{ then } x_1 \text{ else } x_2) \Downarrow_q S_2} 1 \qquad \frac{(\Gamma, y) \Downarrow_v V \quad (\Gamma, x_1) \Downarrow_q S_1}{(\Gamma, \text{if } y \text{ then } x_1 \text{ else } x_2) \Downarrow_q S_1} 2$$

- The semantics of the `for` operator is given in terms of the `For` helper function:

$$\frac{i : \texttt{<avar>} \quad (\Gamma, x_1) \Downarrow_q S_1 \quad h \in \texttt{["sup"|"inf"]}}{(\Gamma, \text{for } i \text{ in } x_1 \ h \ x_2) \Downarrow_q \texttt{For } h \ \Gamma \ i \ x_2 \ S_1}$$

$$\frac{i : \texttt{<avar>} \quad x_2 : \texttt{<query>}}{\texttt{For } h \ \Gamma \ i \ x_2 \ \emptyset \text{ rewrites to } \emptyset} \qquad \frac{i : \texttt{<avar>} \quad ((\Gamma_q, \Gamma_a[i \leftarrow R], \Gamma_g, \Gamma_v), x_2) \Downarrow_q S_2}{\texttt{For } h \ \Gamma \ i \ x_2 \ (S_1 \sqcup \{R\}) \text{ rewrites to } (\texttt{For } h \ \Gamma \ i \ x_2 \ S_1) \square_h S_2}$$

Here we have $R : T_3$, $\Gamma = (\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v)$, $\square_{\text{"}\texttt{sup}\text{"}} = \oplus$ and $\square_{\text{"}\texttt{inf}\text{"}} = \otimes$.

- The `select` operator is treated substituting every instance of "select i from x where y" with "for i in x sup if y then i else empty" which explains the semantics of `select` in terms of the above operators. However a query engine may choose to implement this operator in native mode for performance reasons.

The last `<query>` operator is `property` and depends on the query engine.
In the following rule P is $\texttt{Property } h$ and A_2 is $\{\texttt{Exp } P \ '_1 \ r_1 \ \{e_1, \cdots, e_m\}\}$:

$$\frac{h : \texttt{<refine>}\ \ p_1 : \texttt{<path>}\ \ p_2 : \texttt{<path>}\ \ e_1 : \texttt{<exp>}\ \ \cdots\ \ e_m : \texttt{<exp>}\ \ k \in \texttt{["pattern"]?}\ \ (\Gamma, y) \Downarrow_v V}{(\Gamma, \texttt{property}\ h\ p_1\ \texttt{main}\ p_2\ \texttt{attr}\ e_1, \cdots, e_m\ \texttt{in}\ k\ y) \Downarrow_q \bigoplus\{\{(r_2, A_2)\} | (\exists r_1 \in \texttt{Src}\ k\ P\ V)(r_1, p_1 @ p_2, r_2) \in P\}}$$

When the `main` clause is not present, we assume $p_2 = /$.

Here `Property` h gives the appropriate access relation according to the h flag (this is the primitive function that inspects the RDF graph), see Subsection 2.1.2).

`Src` $k\ PV$ is a helper function giving the source m.s.v. according to the k flag. `Src` is based on `Match`, the helper function handling POSIX regular expressions. Formally:

`Src ""` $P\ V$	rewrites to	V
`Src "pattern"` $P\ V$	rewrites to	`Match` $\{r_1 \mid (\exists p, r2)\ (r_1, p, r_2) \in P\}$ V
`Match` $W\ V$	rewrites to	$\bigcup\{\texttt{Pattern}Ws \mid s \in V\}$

Here `Pattern`Ws is the primitive function returning the subset of W : `SetOf String` whose element match the POSIX 1003.2-1992[16] regular expression `"^"` @ s @ `"$"`.

`Exp` $P\ '_1\ r_1\ E$ is the helper function that builds the group of attributes specified in the `attr` clause. `Exp` is based on `Exp`$'$ which handles a single attribute. Formally:

$f\ P\ r_1\ p_1\ p$	rewrites to	$\{r_2 \mid (r_1, p_1 @ (\texttt{Name}\ p), r_2) \in P\}$	with p : `<path>`
`Exp`$'\ P\ r_1\ p_1\ p$	rewrites to	$\{(\texttt{Name}p, f\ P\ r_1\ p_1\ p)\}$	with p : `<path>`
`Exp`$'\ P\ r_1\ p_1\ (p\ \texttt{as}\ p')$	rewrites to	$\{(\texttt{Name}p', f\ P\ r_1\ p_1\ p)\}$	with p and p' : `<path>`
`Exp` $P\ r_1\ p_1\ E$	rewrites to	$\bigoplus\{\texttt{Exp}'\ P\ r_1\ p_1\ e \mid e \in E\}$	with E : `SetOf <exp>`

When c_1 : `<cons>`, $\cdots, c_n$: `<cons>` and the clause "istrue $c_1, \cdots, c_n$" is present, the set P must be replaced with $\{(r_1, p, r_2) \in P \mid \texttt{IsTrue}\ P\ r_1\ p_1\ C\}$ where C is $\{c_1, \cdots, c_n\}$ and `IsTrue` is a helper function that checks the constraints in C. `IsTrue` is based on `IsTrue`$'$ that handles a single constraint. Formally, if p : `<path>`, $(\Gamma, x) \Downarrow_v V$ and C : `SetOf <cons>`:

$g\ P\ p_1\ p$	rewrites to	$\{r_2 \mid (\exists r_1)\ (r_1, p_1 @ (\texttt{Name}\ p), r_2) \in P\}$
`IsTrue`$'\ P\ r_1\ p_1\ (p\ \texttt{in}\ x)$	rewrites to	$(f\ P\ r_1\ p_1\ p) \wr V$
`IsTrue`$'\ P\ r_1\ p_1\ (p\ \texttt{match}\ x)$	rewrites to	$(f\ P\ r_1\ p_1\ p) \wr$ `Match` $(g\ P\ p_1\ p)\ V$
`IsTrue` $P\ r_1\ p_1\ C$	rewrites to	$(\forall c \in W)$ `IsTrue`$'\ P\ r_1\ p_1\ c$

When the clause "isfalse $c_1, \cdots, c_n$" is present, the set P must be replaced with $\{(r_1, p, r_2) \in P \mid \neg(\texttt{IsTrue}\ P\ r_1\ p_1\ C)\}$ (using the above notation). Note that this substitution and the former must be composed if necessary.

If the `inverse` flag is present, also replace the instances of P in the rule and in the definition of `Src` with $\{(r_2, p, r_1) \mid (r_1, p, r_2) \in P\}$.

[16] Included in POSIX 1003.1-2001: `<http://www.unix-systems.org/version3/ieee_std.html>`.

Expressions denoting a multiple string value

These expressions represent m.s.v.'s (including Boolean values and numbers) and their syntax is described in Figure 2.10 (the start symbol is `<value>`).

The first bunch of `<value>` operators gives the helper functionalities for context manipulation, syntactic grouping and logging. In particular:

- The operators for reading variables and for syntactic grouping:

$$\frac{q : \texttt{<string>}}{(\Gamma, q) \Downarrow_v \{\texttt{Unquote}\ q\}} \qquad \frac{(\Gamma, y) \Downarrow_v V}{(\Gamma, (y)) \Downarrow_v V} \qquad \frac{i : \texttt{<vvar>}}{((\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v), i) \Downarrow_v \Gamma_v(i)}$$

  Here $\Gamma_v(i)$ means $\emptyset$ if i is not defined.

- The logging operator (`stat`):

$$\frac{y : \texttt{<value>}}{(\Gamma, \text{stat}\ y) \Downarrow_v \texttt{VStat}\ \Gamma\ y} \qquad \frac{(\Gamma, y) \Downarrow_v V}{\texttt{VStat}\ \Gamma\ y \text{ rewrites to } V}$$

The second bunch of `<value>` operators deals with m.s.v. management:

- The n-ary union operator and the `count` operator:

$$\frac{(\Gamma, y_1) \Downarrow_v V_1 \quad \cdots \quad (\Gamma, y_n) \Downarrow_v V_n}{(\Gamma, \{y_1, \cdots, y_n\}) \Downarrow_v V_1 \cup \cdots \cup V_n} \qquad \frac{(\Gamma, y) \Downarrow_v V}{(\Gamma, \text{count}\ y) \Downarrow_v \{\#\ V\}}$$

- The `proj` (projection) operator makes a coercion between the types T_4 and T_{0b}:

$$\frac{(\Gamma, x) \Downarrow_q S}{(\Gamma, \text{proj}\ x) \Downarrow_v \{\texttt{Fst}\ u \mid u \in S\}} \qquad \frac{p : \texttt{<path>} \quad (\Gamma, x) \Downarrow_q \{(r_1, A_1), \cdots, (r_m, A_m)\}}{(\Gamma, \text{proj}\ p\ x) \Downarrow_v \texttt{Proj}\ (\texttt{Name}\ p)\ A_1 \cup \cdots \cup \texttt{Proj}\ (\texttt{Name}\ p)\ A_m}$$

$$\texttt{Proj}\ p\ \{G_1, \cdots, G_n\} \quad \text{rewrites to} \quad G_1(p) \cup \cdots \cup G_n(p)$$

  where, for each j such that $1 \leq j \leq n$, $G_j(p)$ means $\emptyset$ if p is not defined in G_j.

The third bunch of `<value>` operators gives the Boolean values support: In particular:

- The test operators (`sub`, `meet`, `eq`, `le`, `lt`):

$$\frac{(\Gamma, y_1) \Downarrow_v V_1 \quad (\Gamma, y_2) \Downarrow_v V_2}{(\Gamma, y_1\ \text{sub}\ y_2) \Downarrow_v (V_1 \subseteq V_2)} \qquad \frac{(\Gamma, y_1) \Downarrow_v V_1 \quad (\Gamma, y_2) \Downarrow_v V_2}{(\Gamma, y_1\ \text{meet}\ y_2) \Downarrow_v (V_1 \between V_2)} \qquad \frac{(\Gamma, y_1) \Downarrow_v V_1 \quad (\Gamma, y_2) \Downarrow_v V_2}{(\Gamma, y_1\ \text{eq}\ y_2) \Downarrow_v (V_1 = V_2)}$$

$$\frac{(\Gamma, y_1) \Downarrow_v \{n_1\} \quad (\Gamma, y_2) \Downarrow_v \{n_1\}}{(\Gamma, y_1\ \text{le}\ y_2) \Downarrow_v (n_1 \leq n_2)}\,1 \qquad \frac{}{(\Gamma, y_1\ \text{le}\ y_2) \Downarrow_v \texttt{F}}\,2$$

$$\frac{(\Gamma, y_1) \Downarrow_v \{n_1\} \quad (\Gamma, y_2) \Downarrow_v \{n_1\}}{(\Gamma, y_1\ \text{lt}\ y_2) \Downarrow_v (n_1 < n_2)}\,3 \qquad \frac{}{(\Gamma, y_1\ \text{lt}\ y_2) \Downarrow_v \texttt{F}}\,4$$

The rules of the same operator are listed in order of precedence (high to low).

The comparisons are extensional an the equality between strings is case-sensitive.

The `eq` operator is introduced because the evaluation of y_1 eq y_2 may be more efficient than that of y_1 sub y_2 and y_2 sub y_1.

As an application of the `sub` and `meet` operators, consider an a.v. set computed by X : `<query>` and a Boolean condition computed by B : `<value>` depending on the variable @u : `<avar>`. The test "is B satisfied for each a.v. in X?" is expressed by "proj X sub proj select @u from X where B" whereas the dual test "is B satisfied for some a.v. in X?" is expressed by "proj X meet proj select @u from X where B".

- The operators about logic (`true`, `false`, `not`, `and`, `or`, `xor`):

$$\frac{}{(\Gamma, \text{false}) \Downarrow_v \texttt{F}} \quad \frac{}{(\Gamma, \text{true}) \Downarrow_v \texttt{T}} \quad \frac{(\Gamma, y) \Downarrow_v \texttt{F}}{(g, \text{not } y) \Downarrow_v \texttt{T}} 1 \quad \frac{(\Gamma, y) \Downarrow_v V}{(g, \text{not } y) \Downarrow_v \texttt{F}} 2$$

$$\frac{(\Gamma, y_1) \Downarrow_v \texttt{F}}{(\Gamma, y_1 \text{ and } y_2) \Downarrow_v \texttt{F}} 3 \quad \frac{(\Gamma, x_2) \Downarrow_v V}{(\Gamma, y_1 \text{ and } y_2) \Downarrow_v V} 4 \quad \frac{(\Gamma, y_1) \Downarrow_v \texttt{F} \quad (\Gamma, y_2) \Downarrow_v V}{(\Gamma, y_1 \text{ or } y_2) \Downarrow_v V} 5 \quad \frac{(\Gamma, y_1) \Downarrow_v V}{(\Gamma, y_1 \text{ or } y_2) \Downarrow_v V} 6$$

$$\frac{(\Gamma, y_1) \Downarrow_v \texttt{F} \quad (\Gamma, y_2) \Downarrow_v V}{(\Gamma, y_1 \text{ xor } y_2) \Downarrow_v V} 7 \quad \frac{(\Gamma, y_1) \Downarrow_v V \quad (\Gamma, y_2) \Downarrow_v F}{(\Gamma, y_1 \text{ xor } y_2) \Downarrow_v V} 8 \quad \frac{(\Gamma, y_1) \Downarrow_v V_1 \quad (\Gamma, y_2) \Downarrow_v V_2}{(\Gamma, y_1 \text{ xor } y_2) \Downarrow_v \texttt{F}} 9$$

The rules of the same operator are listed in order of precedence (high to low).

Notice that `"and"` and `"or"` are evaluated with an early-out (C-style) strategy.

- The `ex` and "dot" operators provide a way to read the attributes stored in avar's.

The `ex` (exists) operator gives access to the groups of attributes associated to the a.v.'s in the Γ_a part of the context and does this by loading its Γ_g part, which is used by the "dot" operator described below.

`ex` is true if the condition following it is satisfied by at least one pool of attribute groups, one for each a.v. in the Γ_a part of the context. Formally we have the rules:

$$\frac{(\forall \Delta_g \in \texttt{All}\ \Gamma_a)\ ((\Gamma_q, \Gamma_a, \Gamma_g + \Delta_g, \Gamma_v), y) \Downarrow_v \texttt{F}}{(\Gamma, \text{ex } y) \Downarrow_v \texttt{F}} 1 \quad \frac{}{(\Gamma, \text{ex } y) \Downarrow_v \texttt{T}} 2 \quad \frac{i : \texttt{<avar>} \quad p : \texttt{<path>}}{(\Gamma, i.p) \Downarrow_v \Gamma_g(i)(\texttt{Name}\ p)}$$

where[17] $\texttt{All}\ \Gamma_a = \{\Delta_g \mid \Delta_g(i) = G \text{ iff } G \in \texttt{Snd}\ \Gamma_a(i)\}$, and $\Gamma = (\Gamma_q, \Gamma_a, \Gamma_g, \Gamma_v)$.

Moreover $\Gamma_g(i)(\texttt{Name}\ p)$ means $\emptyset$ if i or $\texttt{Name}\ p$ are not defined where appropriate.

Here the first rule has higher precedence than the second does.

[17] Δ_g has the type of Γ_g.

```
<attr>  ::= <path> [ "=" <string_list> ]?
<group> ::= "{" <attr> [ ";" <attr> ]* "}"
<av>    ::= <string> [ "attr" <group> [ "," <group> ]* ]?
<avset> ::= [ <av> [ ";" <av> ]* ]?
```

Figure 2.11: Textual syntax of query results

The "dot" operator allows to read an attribute only by specifying an associated avar but this restriction offers the advantage of an unambiguous reference to attributes related to different a.v.'s but sharing the same name. Note that the use of the `let` operator may produce unavoidable attribute name collisions in the scope of nested `where` clauses as in the following example where `...` is a place holder:

```
let %s be property "..." attr "a" of subj "..." in
select @u1 from %s where "..." sub proj
   select @u2 from %s where ex @u1."a" sub @u2."a"
```

2.2.3 Textual syntax and semantics of query results

The textual representations of query result expressions belong to the grammatical production `<avset>` of Figure 2.11, whose semantics is described by four (infix) evaluating relations like the ones used for query expressions. In particular:

- $\Rightarrow_a$: `<attr>` $\rightarrow (T_{0a} \times T_{0b}) \rightarrow$ `Boole` evaluates an attribute.
- $\Rightarrow_g$: `<group>` $\rightarrow T_1 \rightarrow$ `Boole` evaluates an attribute group.
- $\Rightarrow_v$: `<av>` $\rightarrow T_3 \rightarrow$ `Boole` evaluates an attributed value.
- $\Rightarrow_s$: `<avset>` $\rightarrow T_4 \rightarrow$ `Boole` evaluates a set of attributed values.

Note that a multiple string value can be empty. In fact, in an RDF Model a property can be optionally used, even if it is always declared in an RDF Schema.

Also note that the attribute groups are always inhabited.

Formally the evaluation is described by four rules:

$$\frac{p : \texttt{<path>} \quad q_1 : \texttt{<string>} \quad \cdots \quad q_m : \texttt{<string>}}{p = q_1, \cdots, q_m \Rightarrow_a (\texttt{Name}\ p, \{\texttt{Unquote}\ q_1, \cdots, \texttt{Unquote}\ q_m\})} \qquad \frac{z_1 \Rightarrow_a a_1 \quad \cdots \quad z_m \Rightarrow_a a_m}{\{z_1; \cdots; z_m\} \Rightarrow_g \{a_1, \cdots, a_m\}}$$

$$\frac{q : \texttt{<string>} \quad z_1 \Rightarrow_g G_1 \quad \cdots \quad z_m \Rightarrow_g G_m}{q\ \text{attr}\ z_1, \cdots, z_m \Rightarrow_v (\texttt{Unquote}\ q, \{G_1, \cdots, G_m\})} \qquad \frac{z_1 \Rightarrow_v s_1 \quad \cdots \quad z_m \Rightarrow_v s_m}{z_1; \cdots; z_m \Rightarrow_s \{s_1, \cdots, s_m\}}$$

2.3 Some notes on the earlier versions of MathQL-1

This section outlines the differences between the version of MathQL-1 described in Section 2.2 and the earlier versions presented in [Lor02],[Nat02], and [GS03].

MathQL-1.1

The first description of MathQL-1 is briefly sketched in [Lor02] where D. Lordi illustrates his implementation of a query engine for the language. This paper contains only a succinct verbal description of the query expressions without mentioning any formal semantics for them (such a semantics did not exist when the paper was written).
[Nat02] contains a more precise description of the language and includes a sort of formal semantics for it, adapted from [GS03]. However, that semantics is not designed carefully and has some problems: in particular it is not clear how the author can formalize the `sortedby` operator, that sorts the contents of a.v. sets, given that he formalizes a.v. sets using unordered data structures like the `SetOf` type constructor.

That version of the language revealed very unsatisfactory for several reasons:

- a.v.'s have a poorly structured set of attributes (attributes are not grouped and have a single string value) and this causes the major problems in defining a reasonable semantics for the union and intersection of a.v.'s sharing the subject string and some attributes in the case they have different values.
- Relations and functions are built in the language and are optimized for the metadata currently available in the context of the HELM project. This makes the language not usable in other contexts without a suitable syntax extension, which is required even if the HELM metadata model itself happens to be modified.
- The `pattern` clause does not use standard regular expressions but Unix-like wildcards whose semantics is optimized for a very simple URI reference scheme.

MathQL-1.2

The next version of the language, appearing in [GS03] and also included in [Nat02] with some additions taken from the preliminary version of this dissertation (for instance the character escaping syntax), aims at solving these inconvenient in the following way:

- The attributes of a.v.'s are multivalued and partitioned into groups allowing a clean semantics for union and intersection; the group structure justifies the presence of the `ex` operator in connection with the "dot" operator. The use of multiple string values in place of single string values is pushed as far as possible in the language architecture leading to the adoption of other operators like `sub`, `meet`, `eq`, `refof` (now `proj`) and also of variables for values (the vvar's introduced by the `let` operator).
- This version uses abstract (i.e. not built-in) relations and functions whose semantics depends on the query engine, and allows the use of the `inverse` flag as a syntactic support for the inference of inverse relationships.
- [GS03] provides for the use of standard regular expressions in the `pattern` operator.

[GS03] does not include the `sortedby` operator because of the problems it gives in the context of a formal semantics based on unordered structures[18]. Moreover the `fun` operator, coming from [Lor02], is also left out because it is a simplified version of the operator used to invoke abstract functions.

MathQL-1.3

The version of the language presented in this chapter improves the previous version with the aim of satisfying the basic requirements about a query language for RDF metadata listed in Subsection 1.4.1. In particular this version adds:

- More expressiveness in attribute names which are paths instead of identifiers. This choice has a double benefit: on one hand it frees these names from the syntactic constraints of identifiers (which can contain only a limited set of characters) allowing them to encode full RDF property names (including a Namespace when appropriate), and on the other hand it allows attributes to be named as their defining paths, thus making attribute renaming an optional feature in contrast with MathQL-1.2.
- New features for the `property` operator that now includes regular expressions handling (the `pattern` flag), constraint specifications (the `istrue` and `isfalse` clauses) to reduce the computational complexity of some `property` operations, and main component specifications (the `main` clause) to manage structured RDF properties.
- The `add` and `keep` operators as a support for customizing the query results.

[18] The use of unordered structures simplifies MathQL-1 operational semantics quite a lot.

- The `if` and `for` operators, taken from the programming languages tradition (as the `let` operator) help to formulate sophisticated queries.
- New operators for m.s.v.'s (attribute projection, n-ary union, support for numbers) and logging facilities for debugging purposes.
- A better conceived syntax for escaping the characters in constant strings, which aims at complying with the W3C specifications.

In this version of the language, Boolean values (and numbers) are m.s.v.'s. This simplifies MathQL-1 syntax and semantics without any drawback. Currently, we are considering the possibility of removing m.s.v.'s too (in the sense of removing the `<value>` grammatical production) and of using a.v. sets everywhere (i.e. the `<query>` grammatical production) because this would lead to a substantial simplification of MathQL-1 architecture.

In [GS03] the `property` operator presented in Subsection 2.2.2 is called `relation` and another `property` operator is provided. This operator does not appear in our presentation being a particular case of `relation` from the operational standpoint.

Chapter 3

The Hypertextual Electronic Library of Mathematics

In this chapter we present an overview on the HELM project (see also Subsection 1.1.2) which is the framework where we are testing MathQL-1, focusing particularly on HELM metadata structure (see Subsection 3.2.2 and Subsection 3.2.4). Chapter 4 describes how MathQL-1 is exploited by HELM.

The major methodological requirements of the HELM project are:

- **Standardization**. By standardization, HELM only means the representation of the information in a clearly defined, application-independent format. XML is adopted as a neutral specification language to encode the library contents, taking advantage of the many functionalities on XML documents offered by standard commercial tools. Special dialects of XML, such as MathML or XSLT, can be reasonably used as a standardization languages for presentational and notational aspects of the information, or for transformations among XML files.
- **Distribution**. HELM provides a distributed library, which may contain multiple copies of the same document for efficiency and fault-tolerance reasons. For relocation and balancing reasons, the objects of the library have logical names and the system provides a mechanism for resolving these names.
- **Simplicity**. The kernel system should be as light as possible and should profit of the existing technology for Web Publishing. The overall policy should be as liberal as possible, respecting the general philosophy of the Web: every user with a HTTP or FTP space should be allowed to publish his or her piece of theory. In particular,

no assumptions should be made about the server for publishing and users should be able to browse the library with any browser.

- **Modularity**. HELM should be conceived as an open system, in the most general sense of the term. It should be possible to add more functionalities and utilities with a minimal effort and no impact on the kernel.

3.1 The overall architecture

3.1.1 Overview

The HELM project is developed by Prof. A. Asperti and his research team (also including I. Schena, C. Sacerdoti Coen, L. Padovani, F. Guidi, S. Zacchiroli, and formerly D. Lordi, L. Natile and A. Nediani) at the Department of Computer Science of the University of Bologna. The project is integrated with related projects in the framework of the European FET Project IST-2001-33562, under the name MOWGLI[1].

HELM pursues the integration of the tools for the automation of formal reasoning and mechanization of mathematics (mainly proof assistants and logical framework) with the recent technologies for the development of Web applications and electronic publishing.

The main technical novelty of HELM is in its synergy between different scientific communities and research topics as digital libraries [Com98], web publishing and logical environments.

From the web-publishing standpoint, the project is the first attempt to provide a comprehensive description, from content to metadata, of Mathematics, in order to enhance its accessibility, exchange and elaboration through the Web. To this aim HELM exploits most of the technologies recently introduced by W3C like XML, MathML, XSLT, RDF and related companion tools.

From the digital libraries standpoint, the project is aimed at exploiting the functionalities offered by the Web, and in particular a more integrated use of its browsing and searching facilities. In this context, HELM library is not a mere structured collection of texts, but it is a virtual structure inside which the user can freely navigate.

As the library is encoded in XML, HELM needs some modules for exporting the already encoded mathematical knowledge towards the XML representation

[1]Math On the Web: Get it by Logic and Interfaces: <http://www.mowgli.cs.unibo.it>.

Currently, the HELM team has written such a module only for the [Coq] proof assistant but similar exportation functionalities are expected to be provided by the developers of other logical systems in the near future. Note that, while HELM exports the information, a tight interaction with the source application is usually required.

To exploit and augment the library, HELM needs several tools providing the functionalities given by the current tools for proof-reasoning, such as type checking, proof searching, program verification and code extraction. Moreover HELM can use the available well-developed and extensible tools for processing, retrieval and rendering XML-encoded information.

The user will interact with the library through several interfaces that integrate the different tools to provide a homogeneous view of the functionalities. Actually the HELM team is developing two interfaces.

Because of the particular nature of the library, HELM also provides a suitable model of distribution.

3.1.2 HELM vs. other technologies

EULER (see Subsection 1.3.3) can be considered one of the most interesting and representative projects with regard to Digital Libraries. The main difference with the HELM project is that EULER tries to integrate every kind of mathematical resource, while HELM is concerned only with the formal ones. Moreover the format of the resources usually indexed by EULER is textual or presentation-oriented, thus a granular access to the document structure is impossible.

P-MathML is definitely suitable for HELM purposes, but C-MathML does not entirely meet our requirements because it is mainly focused on computer algebra systems, while HELM is oriented towards proof-assistant applications, Also, MathML does not provide a standard semantics for its content elements and this is fundamental in a formal context where semantics is uniquely defined.

Anyway, as we said, OpenMath provides only a semi-formal encoding of mathematical objects, and this fact rises almost the same problems of C-MathML, so under this point of view, OpenMath has no added value than C-MathML for the purposes of HELM.

With regard to MathWeb, which uses the OMDoc format to encode information, we point out the differences with respect to HELM.

- The emphasis of MathWeb is on communication and structured knowledge bases[2] so XML is used to encode short-term persistence information, while HELM is an XML-native system and uses XML encodings also for long-term persistence information.
- MathWeb is mainly focused on computer algebra systems and interoperability issues, whereas, as we said, HELM is concerned with proof assistants.

  The proof format supported by OMDoc allows, as HELM does, natural language representations at every level of abstraction, but it is tactic-based and therefore it is not satisfactory because the language of tactics is really system-dependent, often partly documented and in continuous evolution.

  Furthermore the scripts in this language are usually very obscure to humans because the semantics of tactics is not compositional, depending on the current sub-goal and its environment.

  On the contrary, HELM proof representation is not tactic-based and HELM natural descriptions do not substitute a formal step but simply represent an alternative way of presenting it.
- In the practice of Mathematics the choice of names is essential because names bring a useful and important meaning, often fixed by an ancient mathematical tradition. Moreover, structured names help their mining, facilitate search and retrieve operations and naturally induce a hierarchical organization of metadata, opening the way to simple inheritance mechanisms.

  For these reasons, HELM uses hierarchically structured names for the objects in its library while MathWeb currently relies on flat names even if a different solution is likely to be adopted in the next specification.
- The current effort inside HELM is mostly oriented towards "documents" which are not "theories" in the sense of OMDoc because, at present, any attempt to impose a specific standardization at the level of theories is very likely to be rejected by the scientific community.
- OMDoc allows many different levels of markup (representing content, rendering and metadata information) to be placed in a single document because it is mainly conceived as the input/output format for the MBase system, so documents are very

[2] See for instance MBase: `<http://www.mathweb.org/mbase>`.

complex and it may be difficult to process them efficiently (as it already happened with SGML [ISO8879] in the publishing world).

HELM instead follows the guidelines of W3C, which is moving in the direction of having many dialects, each for a peculiar kind of information, and adopts the general approach of layering the levels of processing,

As a consequence, each different kind of information is kept separated form the other and this leads to a better complexity management because documents are smaller.

- OMDoc metadata describe only the general information about documents by means of the Dublin Core elements because the other information is inferred by the specific distribution server (i.e. MathWeb), while HELM does not make this assumption and needs the full RDF encoding for its metadata.

3.2 The persistent contents of the library

HELM library is currently composed of about 65000 XML documents consisting of *core files* and *auxiliary files*. A **core file** contains a basic unit of information (we will call it an *object*) which is a formal definition, axiom or proven statement expressed in a suitable logical system and typically borrowed from the available repositories produced by proof assistants. Each logical system has a different DTD, due to the different foundational meaning of its operations and constructions. An auxiliary file contains additional information about a core file or about another auxiliary file. The additional information can be of several kinds and includes **RDF metadata**, **theories** (describing the semantic information needed to classify the objects into lemmas, conjectures, corollaries, etc, and to enforce some constraints, for instance, on the scope of variables), **views** (structured collections of references to objects, suitably assembled for some presentational purpose) and **annotations** (storing presentational human-provided information).

In this section we give some details about these kinds of documents.

It is very important to stress that the auxiliary information is essential to enable or facilitate specific functionalities such as rendering, searching and indexing of the core information, but, on the other hand, it may be a burden for an application that processes the core information automatically: either it could be simply ignored or it imposes a lot of additional consistency checks. In this perspective HELM separates the information

according to its meaning and usage storing the core data and the auxiliary data in distinct XML files, which are related using the [XLink] technology.

As a result an application can consult just the XML files holding the information it really needs, without having to parse and ignore non-interesting information.

3.2.1 Objects and theories exported from Coq

The module interfacing HELM with the Coq System is part of Coq version 7 and is used to export Coq's libraries into a suitable low-level XML dialect which is specific to Coq's logical system, i.e. the Calculus of (Co)Inductive constructions (CIC). The exportation procedure is a batch process producing the CIC object files and the associated theory files. The exportation strategy follows a minimalist principle: only the non-redundant information (i.e. the minimal information required for automatic checking) available at the logical level should be exported.

One of the problems connected with the design of the exportation module is that some information which needs to be exported (i.e. the one required for presentational issues) is not directly available from Coq internals. For instance, the type of the inner nodes of a proof[3] (which are essential to recover a human readable representation of the proof: see [CKT95, Cos00]) is typically missing. Note that the information about these types is kept separate from the core information for the reasons we explained above.

Each CIC object file contains an XML representation of a CIC object that can be a definition (also inductive or coinductive), a proof in progress (i.e. an unfinished proof), an axiom, or variable declaration. Note that a definition can actually be a theorem because CIC does not make any syntactic distinction between these two concepts.

Each object is encoded with its type and its body (when appropriate) which are both CIC terms. These files are hierarchically structured into directories corresponding to sections in Coq (used for instance as delimiters of the scope of a local axiom or definition) and for each leaf directory a theory file is also extracted from Coq internals.

The theory file is an XML document describing the objects in that directory, with the aim of assigning a semantic classification to them (the distinction between definition and theorem is done at this level) which does not depend on the specific logical framework. A theory does not include its objects directly, but refers to them trough their URI's.

[3]Recall that CIC represents propositions as types, following the so-called Curry-Howard isomorphism.

It is important to stress that at the theory level, objects are not organized in chapters, paragraphs or the like because many kinds of markup languages have just been developed to do so. For this reason, and accordingly to the spirit of XML, the theory markup is freely mixable with other kinds of markup, such as [XHTML].

3.2.2 Intrinsic and extrinsic RDF metadata

In the context of the HELM project, the purpose of metadata is to provide semantic information about the library contents enabling smart searching and retrieving functionalities (see also [Ric99]). These features, besides being very helpful to browse the library, are also fundamental in the development of those proof assistants meant to allow an effective reuse of already developed results. It is a matter of fact that many theorems and definitions provided by Coq or by other similar tools are often restated by the authors in the new contributions for the mere difficulty of identifying the needed notion in the already developed knowledge base.

Note that, as a general-purpose metadata model (like the Dublin Core metadata) does not suffice to describe the particular semantics of mathematical information, HELM needs to use a specialized model that we will discuss hereafter. In particular HELM metadata can be associated either to a whole document (i.e. object or theory) like for instance the author of a theorem, or to some part of it (i.e. an informal description of a proof step).

HELM distinguishes between two kinds of metadata: intrinsic and extrinsic. By intrinsic metadata we mean all the information that can be automatically recovered by analyzing the documents contents (such as the list of identifiers occurring in a term, or the main identifier in the conclusion of a statement). Extrinsic metadata are the auxiliary data that cannot be inferred from the documents themselves (such as the name of the author, the date, or a list of keywords).

Intrinsic metadata are generated in a completely automatic way by a batch process while extrinsic metadata, including the Dublin Core and EULER elements, are manually inserted using suitable tools.

In Subsection 3.2.4 we describe the latest version of HELM RDF Schemata respectively for theories and objects[4] Which are entirely due to I. Schena [Sch02]. These Schemata

[4]An exhaustive presentation should include the Dublin Core and EULER elements, but here we are taking their Schema hierarchies for granted.

include both kinds of metadata and are still under development (as the whole surrounding technological infrastructure). Currently we are starting to test the corresponding Models against concrete requirements.

HELM Schemata provide for the following features:

- **XML Schema data types support.** The literal values of properties, i.e. the range constraints, are specified using the XML Schema data types [XMLS] as defined in the XMLSchema-datatypes RDF Schema ("dt" Namespace)[5] because it is very likely that the RDF working group will propose the usage of these types for the specialization of the *rdfs:Literal* class.

- **Dublin Core metadata support.** HELM exploits a corrected version of the RDF Schema for the Dublin Core Element Set 1.1[6] ("dc" Namespace), a corrected version of the RDF Schema for the Dublin Core Element Set Qualifiers [DCT] vocabulary[7] ("dcq" Namespace) and a corrected version of the Dublin Core Schema for the type vocabulary[8] ("dct" Namespace). All DC properties must apply to *hth:MathResource*, representing a mathematical resource in general.

- **Euler metadata support.** HELM uses a corrected version of the RDF Schema[9] provided by the SCHEMAS project[10]: a forum for metadata Schema designers involved in projects under the IST Programme and national initiatives in Europe.

3.2.3 Annotations

By annotations we mean comments, notes, explanations, or other types of external remarks that can be attached to any Web document or to a selected part of it without modifying the document itself. The idea is that a user getting a document, may also load its annotations from some selected annotation servers. From the technical standpoint, annotations are usually considered as metadata, since they give additional information about an existing piece of data, even if this information is not always machine-understandable.

[5] See: `<http://www.w3.org/2001/XMLSchema-datatypes>`.
[6] Original at `<purl.org/dc/elements/1.1/>`, corrections at `<helm.cs.unibo.it/schemas/dces>`.
[7] Original at `<purl.org/dc/terms/>`, corrections at `<helm.cs.unibo.it/schemas/dcq>`.
[8] Original at `<purl.org/dc/dcmitype/>`, corrections at `<helm.cs.unibo.it/schemas/dctype>`.
[9] `<wip.dublincore.org/2000/11/21-euler>`, corrections: `<helm.cs.unibo.it/schemas/euler>`.
[10] See `<http://www.schemas-forum.org/>`.

The Annotea project[11] [KKPS01] is part of W3C Semantic Web activity and it is meant to enhance the W3C collaboration environment with shared annotations. Annotea describes annotation with an RDF based annotation Schema and exploits [XPointer] for locating the annotations in the annotated document. These annotations can be roughly considered as attachments containing plain text associated to a (fragment of) document to make it understandable, are stored on annotation servers in RDF format and are presented to users by suitable clients that interface with the servers using the HTTP protocol. The first client implementation of Annotea is the W3C's Amaya MathML compliant editor and browser[12].

HELM annotations are different because they are meant to give an alternative human-readable view of a formalized entity as a CIC term (which can be a whole proof) in a core file. In this setting, the annotation content is not required to be machine-meaningful and is encoded in plain XML. Every entity has a unique identifier in the file where it is stored and these identifiers are used to link the annotations to the entities (this technology was found to be more convenient than the use of XLink).

In any case, as in Annotea, HELM annotations aim at making information human-understandable, so this can still be regarded as metadata.

3.2.4 The RDF Schemata for metadata about theories and objects

Metadata about theories allow to search for theory items belonging to a given semantic category (axiom, variable, definition, theorem, lemma, corollary, fact, data, type, algorithm) and to explore various kinds of dependencies among theory items.

A stable (but reduced) RDF Schema for metadata about theories ("hth" Namespace) is available at `<http://helm.cs.unibo.it/schemas/schema-helmth>` and includes the following items. The full development version of this schema is available at: `<http://helm.cs.unibo.it/¨schena/schema-hth>`.

- **MathResource**. The subclass of *rdfs:Resource* representing mathematical resources described in the HELM library by core XML documents. All properties referring to core documents use this class as the domain constraint so every subclass of *MathResource* can naturally inherit them.

[11] See: `<http://www.w3.org/2001/Annotea/>`.

[12] See `<http://www.w3.org/Amaya/>`.

In particular *MathResource* is the domain of the following properties:

  - **shortName**. The subproperty of *dc:title* describing the short name (i.e. alias) of the mathematical resource. The value of this property is a *dt:string*.
  - **firstVersion, modified**. The subproperties of *dc:description* describing any additional information about the first and the modified version of the mathematical resource. The value of these properties is a *dt:string*.
  - **institution, contact**. The subproperties of *dc:creator* describing the information about the affiliated institution and the contact of the creator of the mathematical resource. The value of *institution* is a *dt:string* while the value of *contact* is an instance of *Contact* (see below).
  - **isBasedOn, isBasisFor, isSourceFor, hasSource**. The subproperties of *dc:relation* describing the respective relationships between mathematical resources. Their value is a *MathResource*.

- **Theory**. The subclass of *MathResource* representing mathematical theories.
- **HELMFormat**. The instance of *dcq:FormatScheme* representing the logical encoding format of a mathematical resource. Possible values (specified in its *rdf:about* attribute) may include "XML.cic", "XML.mizar" and "XML.hol".
- **HELMId**. The instance of *dcq:IdentifierScheme* representing the identifiers of a mathematical resource.
- **HELMText**. The subclass of *dcq:Text* representing the text types of a mathematical resource. Possible values include: "Abstract", "Paper", "Bibliography", "Enclosure", "HomePage", "LectureNotes", "Monograph", "PatentSpec", "Preprints", "Proceedings", "Review", "Separatum", "Serial", "TechReport" and "Thesis", plus the entry "General" that means: none of the former.
- **HELMSoftware**. The subclass of *dcq:Software* representing the software types of a mathematical resource. Possible values include: "Exec" and "Source".
- **Contact**. The class representing the creator contact information. It is the domain of the following properties whose value is a *dt:string*:
  - **address, email**. The properties describing the respective contact information relative to the creator of a mathematical resource.

Metadata about objects can describe the structure of a HELM object (i.e. a mathematical statement or proof) in terms of its constants and variables taking into account their position that is classified in semantically meaningful categories. Note that the constants include the names of the referred objects so these metadata also describe the dependencies between objects.

Other metadata can describe proofs (which, being actually terms, are treated at the object level) by means of their proof steps (using tactics or proof-plans[13]) taking into account the information about their logical foundation (i.e. intuitionistic, classical, predicative, impredicative, based on contested axioms, etc.)

A stable (but reduced) RDF Schema for metadata about objects ("h" Namespace) is available at `<http://helm.cs.unibo.it/schemas/schema-helm>` and includes the following items. The full development version of this Schema is available at: `<http://helm.cs.unibo.it/¨schena/schema-h>`.

- **Object**. The subclass of *hth:MathResource* representing HELM units of knowledge. The elements of this class inherit the properties of a *hth:MathResource* (for instance *hth:shortName* and the Dublin Core properties), but note that inheritance is a delicate issue since an object can be referred by several theories.

  object is the domain of the following properties:

  - **refObj**. The property describing a referred constant. Its value is an anonymous resource with the properties *occurrence* (describes the referred object), *position* and *depth* (see below).
  - **backPointer**. The property describing a reference to the object (i.e. the inverse of the former). Its value is an anonymous resource with the properties *occurrence* (describes the referring object), *position* and *depth* (see below).
  - **refRel**. The property describing a referred variable. Its value is an anonymous resource with the properties *position* and *depth* (see below).
  - **refSort**. The property describing a referred CIC sort. Its value is an anonymous resource with the properties *sort*, *position* and *depth* (see below).

  The properties representing the components of a reference description are listed below:

[13] In the sense of OMDoc.

  - **occurrence**. The property describing an object involved in a reference. Its value is an **Object**.
  - **sort**. The property describing a referred CIC sort. Its value is an instance of *Sort* (see below).
  - **position**. The property describing the position of a reference in the referring object. Its value is an instance of *Position* (see below).
  - **depth**. The property describing a depth index associated to the position of a reference in the referring object. The *depth* of a reference is defined only when its *position* is set to *MainHypothesis* or *MainConclusion* (see below). The value of *depth* is a *dt:integer*

- **DirectoryOfObject**. The subclass of *hth:MathResource* describing the hierarchical collections (directories) of objects. A collection contains the objects whose URI's share a common URI prefix (which identifies the collection itself and is placed in the *rdf:about* attribute of *DirectoryOfObject*). Note that the objects of a collection may share some DC and EULER properties.

- **Position**. The class representing the positions of a reference in the referring object.

- **MainHypothesis, InHypothesis, MainConclusion, InConclusion, InBody**. The instances of *Position* representing a classification of mathematically meaningful positions for a reference in the referring object. The positions include: "in head position of a statement premise", "in a statement premise, but not in head position", "in head position of the statement conclusion", "in the statement conclusion, but not in head position" and "not in the statement".

- **Sort**. The class representing the possible CIC sorts ("set", "prop", "type").

- **Set, Prop, Type**. The instances of *Sort* representing the corresponding CIC sort.

3.3 The processing tools

This section briefly describes the software tools implemented by the HELM team to process the persistent contents of the library. HELM architecture allows every user to consult and contribute to the library requiring as few client-side software as possible. In particular, at least for simple consulting, HELM proposes a Web interface requiring

only a common browser. The on-line version of the library can be found at the address <http://helm.cs.unibo.it/library.html>. A plug-out to render MathML documents and another one to annotate proofs can be freely downloaded. The Web interface provides access to an integrated proof checker running on the server side.

3.3.1 The render engine

In HELM the transformation of a document from its persistent XML encoding described previously in Section 3.2 to its rendering encoding, essentially consists of two phases: *abstraction* and *presentation*.

Both the abstract format and the rendering format are generated on the fly by means of XSL Transformations [XSLT]. Most of these transformations are pretty complex: HELM heavily relies on the XSLT inclusion mechanism to organize stylesheets in a coherent and easily maintainable structure.

Furthermore there are several flows of transformations: one applies to individual objects, another one applies to theories. The most complex transformation process obviously concerns objects and their sub-elements (i.e. CIC terms interpreted as propositions or proofs, requiring an adequate notational support) while theories are essentially structured collections of references to objects and do not require major transformations.

In the **abstraction phase** the semantic content of the information is extracted from its logical encoding and expressed in a suitable content-centric XML dialect, while a back-pointer to the logical encoding is always preserved as an XLink.

The content level is meant to improve the modularity of the whole architecture and here many different formal notions, from the same or even from different logical environments, are typically mapped into the same content notion (think for instance of the notion of equality whose formal definition may vary from one system to another, but whose intended meaning does not change). So, there is no point in defining a specific presentation for each formal notion at the logical level and HELM just defines presentation for the content level.

HELM adopts C-MathML (see Subsection 1.2.1) for the content representation of formulae and proofs, and has defined a new markup for the theory and metadata level.

During abstraction, HELM must work heavily on proofs in order to put them in a suitable form for human reading. Typically this requires a major reorganization of the structure of the proof (see for instance [CKT95, Cos00]): in proof assistants, proofs are

typically generated in a top-down fashion while we naturally expect a bottom-up presentation where sub-proofs appear before conclusions. Another issue is that of recognizing and managing induction principles, (one of the main proof-mechanisms of constructive mathematics). During this phase, proofs are also integrated with their "inner types" (i.e. intermediate conclusions). One of the most interesting achievements of HELM has been to prove that even these complex transformations can be feasibly performed using XSLT.

In the **presentation phase** the document is turned from its content representation to its final rendering format (currently HELM produces either P-MathML or [XHTML]; other languages could be exploited in the future). This transformation is based on a bunch of pre-defined or user-defined XSLT stylesheets containing notational and stylistic intelligence. HELM uses, among others, a stylesheet by Igor Rodionov[14] which is compliant with the last MathML specification. Most of these stylesheets have a very simple and repetitive structure, so the HELM team is currently studying the possibility of generating them automatically form a more abstract and concise representation of notational and stylistic information. For example most of the rules dealing with operators of the same arity are similar and could be inferred from the arity, the associativity, the type (infix/prefix/postfix) and the precedence index of the operators themselves.

The XHTML rendering format is provided because no already available browser fully implements the MathML specification and behaves correctly for HELM quite peculiar documents (for dimensions and level of table nesting). Moreover, hyperlinks must be added to MathML documents only using XLink, therefore HELM requires browsers that are both MathML and XLink compliant. Moreover, even if we can expect such browsers to be developed very soon, it is quite unlikely that they will have the possibility of nesting different markups (i.e. XHTML and MathML) in the same document; note that this feature is needed to render both mathematical documents and user-annotated formal proofs.

The main issues that can not be easily addressed regarding XHTML rendering are:

- **Rendering oversized formulas**. A MathML engine has enough information to correctly break the lines that exceed the page width re-indenting the output accordingly. For XHTML, HELM lets the browser create an oversized canvas. Note that the presentation stylesheets already do a good job computing a coarse-grained layout so the event of oversized formulas is very rare and not very severe.

[14] Current home page: `<http://www.orcca.on.ca/MathML/igor.html>`.

- **Multiple-depth formula rendering**. A MathML presentation element, named *maction*, is used to create a node having many children of which only one is shown, and the user can switch the visible one. HELM exploits this element allowing to browse a proof as a collapsing tree where the user can expand the proofs in a progressive way, augmenting the level of detail only locally. This can not be reproduced in XHTML and to achieve a similar effect, HELM can simply render the whole proof again with different parameters, but this operation requires many seconds if the proof is huge.
- **Smart selections**. A MathML render engine can easily allow to select subexpressions in a semantically meaningful way. Even if it is unlikely that standard browsers could be exploited for complex interactions (i.e. editing), this feature seems to be very helpful in order to understand the structure of complex expressions.

3.3.2 Other tools

HELM provides for a variety of tools for handling the library contents. Here we give an overview of the most important ones[15]:

- **The tools for rendering MathML.** As a compromise to cope both with the limitations of XHTML and the unavailability of MathML compliant browsers (recall that Amaya and Mozilla support only a subset of MathML, and that their performances are quite low), HELM provides a plug-out for Netscape under Linux, the GtkMathView[16], with rendering[17] and interaction capabilities for documents embedding MathML presentation markup. The widget adopts Gdome2 [CP02], a DOM level 2 Gnome implementation that is another by-product of HELM. Using Gdome2, it will be possible to integrate this widget with other engines with the final aim of developing an architecture in which different kinds of markup will be intermixed in the same document and will be rendered by co-operating widgets.

  The plug-out controls Netscape in such a way that the browser and the plug-out windows are kept in synch. This means that the XHTML page always refers to

[15]The tools for rendering MathML are maintained by L. Padovani while the annotation software, the type-checker and the proof engine are maintained by C. Sacerdoti Coen.

[16]see: `<http://helm.cs.unibo.it/mml-widget>`.

[17]Both to screen and Postscript.

the object shown in the plug-out: i.e. following a hyperlink in the plug-out or in the browser, both documents are updated. That is important because the XHTML page holds a control frame that proposes the actions allowed on the current object (for instance annotate it or type-check it).

In this way, the control frame and its JavaScript logic have not to be reproduced in the HELM plug-out, that is kept extremely simple. This solution, though, has some limitations too:

- It requires the user to install client-side software. Moreover it works only for Linux boxes and, without modifications, only with Netscape Navigator.
- Due to the way the plug-out is kept in synch with Netscape, progressive rendering is not allowed. Hence the user has to wait for the entire file to be downloaded before Netscape gives it to the plug-out for rendering.
- The plug-out solution does not work for MathML embedded inside other kinds of markup.

- **The annotation software.** If we leave the problem of consulting the library and we focus on more advanced forms of interaction, for example annotating formal objects with informal descriptions, the only feasible solution seems to require the user to install client-side software.

  HELM provides a plug-out for annotating CIC terms, which is invoked by following a link in the control frame of the interface. The link returns a document with a suitable MIME type that causes the browser to start the plug-out. Here HELM is implicitly assuming that downloading an applet every time we need it is too time-consuming. The reason is that the applet should be at least able to render MathML and this requires a huge amount of byte-code to be downloaded even for simple objects.

- **The proof checker.** HELM comes with a stand-alone proof checker for CIC objects, which works directly on the XML logical encoding. It is similar to the one used by Coq but fairly simpler and smaller thanks to its independence from the proof engine (see below). With respect to other proof checkers (as the one of Coq), it is fairly standard except for the peculiar management of the environment: usually each checked object is added to an environment and it is used in subsequent checking, so every object is always checked with the same, statically defined environment.

The HELM proof checker, instead, builds the environment (a cache, actually) "on-demand": every time an object that is not in the environment is referred, the current proof checking is suspended and the new object is checked. This process is recursive and checks are introduced in order to avoid cyclic references.

- **The proof engine.** HELM proof engine is meant as an authoring tool exploiting the all set of functionalities offered by HELM architecture. This tool is accessed through a graphical front-end, the claims to prove are either typed using a Coq-like syntax, or retrieved from the library typing their HELM URI. Every incoming object is type-checked and objects are output using the render engine (Subsection 3.3.1). The proof construction is handled by tactics, which are invoked using some graphical controls.

  Currently the proof engine implements most of Coq tactics plus a tactic exploiting the retrieving capabilities offered by MathQL-1 queries. In particular we would like to mention the following tactics:

  Ring, Fourier.

  The first allows to replace, in the goal, equal terms belonging to a commutative semi-ring structure. The second allows to solve linear inequalities on real numbers using Fourier's method[18].

  SearchPattern_Apply.

  Finds the theorems that can be applied to the goal, among the ones provided by the library, and lets the user select which theorem to apply. This tactic explores the library issuing a MathQL-1 query to be detailed in Subsection 4.1.3.

3.3.3 Implementation issues

The main requirement for the HELM user interface is that any user with a Web space (either HTTP or FTP) should be able to browse the library and contribute to it using a minimal amount of client-side software. The reason for this approach is that often the Web space of a user is hosted by a provider that does not allow new software to be run in it. Moreover, the HTTP publishing model has already proved itself very effective in creating distributed libraries of knowledge.

[18] J.B.J. Fourier. *Fourier's method to solve linear inequations/equations systems.* Gauthier-Villars, 1890.

Furthermore the HELM distribution model regards formal mathematical documents as immutable resources in the sense that each new version of a document is considered a different document that does not replace the old version (as for packages of operating systems distributions).

HELM documents are identified by logical names (URI's) instead of physical names (URL's) so different servers can publish the same document with the same name and a user is free to access this document from the nearest or less overloaded server. Using this policy, HELM allows users to make local copies of the documents they are interested in, and to start distributing them. Thus interesting documents are likely to increase the number of their instances avoiding the danger of disappearing.

These are the main problems concerning this model which are still unsolved:

- HELM needs a well-conceived naming policy that does not allow different documents to be published with the same URI (i.e. logical name). This issue can be faced by means of a centralized naming authority even if a distributed solution would be more desirable (for instance a name server hierarchy).
- HELM needs to define how a user can locate and download an instance of a document knowing only its logical name and a list of servers possibly providing it.
- HELM needs to define how a user can locate the sites that are able to process the documents (i.e. for rendering or other purposes).

HELM architecture requires at least three components, which are: distribution sites, standard browsers with plug-outs, and processing sites (providing XSLT processors to elaborate the documents, query engines to search the library and others). Distribution sites are HTTP and FTP servers while browsers are HTTP clients running on the user host. Processing sites are HTTP servers providing specific services in response to a browser request. It may happen that processing sites need to contact other sites in order to obtain some information (as a document published by a distribution site) so these components can be HTTP clients too. Note that each HELM component can run on a different host.

The above considerations show that HELM architecture is organized as a set of HTTP pipelines where each node is seen as an object providing different methods taking arguments: a different URL is associated to each method and its *search part* (i.e. the one after the question mark) is used to pass arguments in the standard way.

Here are some comments on the main components:

- **Distribution sites.** Each distribution server publishes a list of the URI's of its documents with the associated URL's (whose transfer protocol can be HTTP, FTP or NFS) and the file format (due to the high verbosity of XML files, these can be stored in compressed form).

- The **Getter** is the client module of the distribution sites and its main method takes an URI and returns an instance of the corresponding HELM document (always in deflated format). Another method allows to specify an ordered list of servers which the Getter contacts on a regular basis, retrieving the lists of available documents and building a local table (a NDBM database) with the URI-to-URL mappings.

  Being supposed to reside closer to the user than the distribution site, the Getter implements a cache that reduces the downloading time of already retrieved documents.

- **UWOBO**[19] is an XSLT processor whose main method is used to apply a list of stylesheets (each one with the respective parameters) to a document. Both the stylesheets and the document are identified by URL's so they can reside on any host. In particular, the document URL is usually the invocation of the *Getter* method to download a document whose URI is also given in the dynamic part of the URL. UWOBO can process precompiled stylesheet to improve performance.

- **hbugs** is a set of components, maintained by S. Zacchiroli [Zac03], generating run-time "suggestions" about the applicable tactics, during an interactive proof. This set includes a *broker*, a *client* and some *tutors* (one for each "suggestion"). The set of "suggestions" is user-defined and the architecture resembles that of Ω-ants [BS02].

- The **query engine** provides a method to execute a general MathQL-1 query (encoded in textual syntax as an argument) plus other methods to execute specific MathQL-1 queries like the one used by the *SearchPattern_Apply* tactic of the HELM proof engine (see Subsection 3.3.2). this component will be analyzed in Subsection 4.1.3.

HELM components used to be implemented in various programming languages (Java, Perl, C/C++ and others) but now the team is strongly oriented towards a uniform [Caml][20]

[19] See: `<http://helm.cs.unibo.it/uwobo>`.

[20] A strongly-typed functional language from the ML family. `<http://caml.inria.fr>`.

re-implementation of the whole software. In this context some Caml bindings are used to interface third party libraries.

Chapter 4

The use of MathQL-1 in the HELM project

In this chapter we describe how HELM (see Chapter 3) exploits MathQL-1 (see Chapter 2) to query the metadata about its library (see Subsection 3.2.4) and the achieved results. Our contribution is this sense consisted in implementing and testing the necessary software components. The integration of MathQL-1 in HELM is still at an early stage but will evolve quite quickly because HELM seems to be a good test case of our query language.

4.1 The MathQL-1 Suite for HELM

The MathQL-1 Suite for HELM includes the software components used by HELM to exploit the features of MathQL-1. Each component is implemented in Caml for an easy integration inside HELM architecture and in the following we will assume that the reader has some knowledge of this programming language.

4.1.1 The basic Caml package for MathQL-1

The first task of a Caml implementation of MathQL-1 is to provide a Caml *package* allowing an access to:

- A representation of MathQL-1 basic structures, which are queries and query results.
- A set of functions enabling the translation between the Caml representation of these entities and their linearized representation in text (see Subsection 2.2.2 and Subsection 2.2.3) and XML.

This is the aim of the basic MathQL-1 package that is maintained by us and that contains two modules, `MathQL` and `MQueryUtil`, which do not depend on HELM architecture.

```
val text_of_query  : (string -> unit) -> MathQL.query -> string -> unit
val text_of_result : (string -> unit) -> MathQL.result -> string -> unit
val query_of_text  : Lexing.lexbuf -> MathQL.query
val result_of_text : Lexing.lexbuf -> MathQL.result
val xml_of_query   : (string -> unit) -> MathQL.query -> unit
val xml_of_result  : (string -> unit) -> MathQL.result -> unit
val query_of_xml   : Lexing.lexbuf -> MathQL.query
val result_of_xml  : Lexing.lexbuf -> MathQL.result
```

Figure 4.1: The Caml Interface for the conversion functions

- The `MathQL` module contains the definitions of the Caml types used to represent queries and query results: The two main definitions are self-explanatory:
  ```
  type query  = ...
  type result = ...
  ```
  The full definition is included in Subsection A.1.1. Note that the constructor for the `ex` operator is declared with two arguments by means of the line:
  ```
  Ex of avar list * msval
  ```
  where the first argument is a list of avar's that does not appear in MathQL-1 syntax. This list is included only for optimization purposes and contains the avar's actually involved in "dot" operators inside the m.s.v. expression `ex` applies to (i.e. its second argument in the above declaration). Knowing this list in advance, which usually has less elements than the whole Γ_r component of the evaluation context Γ (see Subsection 2.2.2), a MathQL-1 interpreter can reduce the size of the set `All` Γ_a involved in the computation of `ex`, and thus it can gain in efficiency.

  Note that this list should not contain duplicate elements.

  Also note that the `MathQL` module does not include a declaration for the context Γ, which is left to each specific interpreter. This choice is justified by the fact that there are many sensible ways to implement the notion of an *association set* (see Subsection 2.2.1).

- The `MQueryUtil` module provides eight functions, shown in Figure 4.1, converting the different representations of queries and query results. Currently, these functions convert the textual and XML syntax into the Caml representation or vice-versa. The names are self-explanatory and the types may be subjected to small changes.

  The first argument of the rendering functions is a call-back of type `string -> unit` connecting them to the output device. The third argument of `text_of_query` and

`text_of_result` is an optional separator used for presentational purposes (normally a new-line character or a `<P>` HTML tag), which is inserted after the linearization of a query or after the linearization of every a.v. in an a.v. set.

Note that `text_of_query` and `xml_of_query` hide the extra argument of the `Ex` constructor, while their inverse functions build it automatically.

The Caml representation of a MathQL-1 constant string is a term of type *string*, but Caml strings are made of ISO 8859-1 characters, whereas MathQL-1 constant strings are made of Unicode characters. So the parsers invoked by `query_of_text`, `result_of_text`, `query_of_xml` and `result_of_xml` map every Unicode character outside ISO 8859-1 to U+001A[1] when it appears in a constant string[2].

Conversely `text_of_query` and `text_of_result` escape every character outside the closed interval U+0020 to U+007E, when appearing in a constant string.

Finally, the parsers invoked by `query_of_text` and `result_of_text` understand (possibly nested) Caml-like comments (i.e. strings between the tokens "(*" and "*)").

4.1.2 The MathQL-1 interpreter for HELM

The second task of a MathQL-1 implementation is to provide an interpreter for the language (i.e. a query engine). The current MathQL-1 formal model (see Subsection 2.1.2) does not specify how the query engine should obtain the information it needs to compute the access relations, so a part of the engine has to depend on the specific metadata it is going to query (we are trying to reduce this part as much as possible, and eventually we will eliminate it). On the contrary, the part of the engine implementing the semantics of every operator other than `property`, should not depend on these metadata.

The Caml package implementing the interpreter for MathQL-1.1 was initially maintained by D. Lordi [Lor02], The upgrade to MathQL-1.2 was due to L. Natile [Nat02], and we are personally maintaining the MathQL-1.3 upgrade that is a full implementation of the language described in this dissertation.

This section describes the current state of the interpreter focusing on two aspects: how the metadata are accessed (the back-end) and how they are made available (the front-end).

[1] This behaviour agrees with the Unicode specification.

[2] A Java representation of MathQL-1 structures would not have this drawback because Java is based on Unicode characters.

According to the HELM team present experience, the fastest way to access the metadata is to parse the RDF Model files describing it, build a relational database containing the collected information, and then let the interpreter access this database. This technique, which is used by many RDF query engines (see Subsection 1.4.2), has the advantage that the RDF files are parsed only when the database is built so the interpreter can access already parsed information, stored in a convenient way. The drawback, however, is that the database needs to be updated whenever new metadata is available (in the specific case of HELM this happens every time users contribute to the library creating new objects).

In this perspective, HELM maintains a relational database managed by the PostgreSQL[3] DBMS which stores all the metadata coming from HELM RDF files (see Subsection 3.2.2). This database is filled by a batch process that scans the metadata files and currently is organized as follows:

- The Dublin Core information is stored in 15 tables named as the corresponding property (i.e. "dctitle", "dccreator", "dcsubject", etc.) and each table has two columns named "uri" and "value". The entries of the first column can be replicated to indicate that a resource defines more than one value for a given property.
- The same approach is used to store the information about the *hth:shortName* property (see Subsection 3.2.4).
- The *h:refRel* property (see Subsection 3.2.4) uses a three column table: the columns are named "uri", "position", "depth".
- The *h:refSort* property (see Subsection 3.2.4) uses a similar table but with an additional column named "sort".
- The information about the *h:refObj* and *h:backPointer* properties (Subsection 3.2.4) is stored in a two-level structure: a table named "registry" associates a unique numeric identifier to each HELM URI (say 1, 2, 3, etc.) and there are other tables (named "t1", "t2", "t3", etc) that contain the metadata about the resource related to the corresponding identifier. These columns of these tables are named "prop_id", "uri", "position", "depth". In each record, "prop_id" discriminates between *h:refObj* and *h:backPointer* information, holding "F" or "B" respectively, "uri"

[3]See `<http://www.postgresql.org>`.

comes from *h:occurrence*, and "position" holds one of "MainHypothesis", "InHypothesis", "MainConclusion", "InConclusion", "InBody" each prefixed by: "http://helm.cs.unibo.it/schemas/schema-helm#"[4].

This database structure was proposed by D. Lordi that in [Lor02] proves it to be much more efficient than the one used by ICS-FORTH RSSBD (see Subsection 1.4.2) with the same metadata.

Another (slower) way to access the metadata is having the interpreter read the RDF Model files at run-time every time a piece of information is needed. In this perspective, the interpreter maintained by L. Natile can also read the HELM RDF Model files using Galax[5], which is a Caml implementation of a fully compliant XQuery engine.

This feature has not been upgraded to MathQL-1.3 yet but we hope to do it very soon.

The other interesting aspect of the interpreter concerns the way it makes the metadata available in MathQL-1 queries. In this sense the access relations recognized by our interpreter are defined applying the general metadata encoding and accessing principles of MathQL-1 (see Subsection 2.1.2) to the HELM RDF graph (see Subsection 3.2.4).

Our interpreter currently supports just one access relation concerning the (possibly compound) RDF properties whose domain is *hth:MathResource* or a subclass of its (i.e. *h:Object*). The other access relations, taking into account the RDF property hierarchies and the other notions inferred from RDF Schema, are not implemented at the moment because the RDF Schema graph used by HELM metadata is not very sophisticated.

The triples of the implemented access relation concern the following properties:

- The unstructured, not compound properties (i.e. *hth:shortName* and the Dublin Core ones) are accessed through paths having the corresponding name (the names are actually those of the corresponding tables in the HELM relational database). So these paths are named `"hth:shortName"`, `"dc:title"`, `"dc:creator"`, `"dc:subject"`, ...
- The unstructured compound properties are accessed through paths having two components. For instance the paths for *h:refObj* are named `"h:refObj"/"occurrence"`, `"h:refObj"/"position"` and `"h:refObj"/"depth"` while the other ones, concerning *h:backPointer*, *h:refRel* and *h:refSort*, are similar[6] (see Subsection 3.2.4). Note

[4]The Namespace of the *schema-helm* RDF Schema (see Subsection 3.2.4).

[5]See `<http://db.bell-labs.com/galax/>`.

[6]We are planning to introduce the "h:" Namespace in the second component of these paths too.

```
val execute : string -> MathQL.query -> MathQL.result
val init    : string -> unit
val close   : string -> unit
val check   : string -> bool
```

Figure 4.2: The Caml Interface for the interpreter

that the values of `"h:refObj"/"occurrence"` and `"h:backPointer"/"occurrence"` are HELM URI's.

Also note that the two-level structure of the HELM database does not allow a simple inversion of the triples of this group concerning *h:refObj* and *h:backPointer*.

- The structured properties are accessed through paths having the corresponding name. Note that the HELM RDF Models do not define any default main component for these properties (this would by done using *rdf:value*).
  - `"h:refObj"`, `"h:backPointer"`.
    The components of their value (i.e. *h:occurrence*, *h:position* and *h:depth*) are reached using the paths `"occurrence"`, `"position"` and `"depth"`. Note that by the definition of *h:refObj* and *h:backPointer*, we have that `"h:backPointer"` is the inverse of `"h:refObj"` (in MathQL-1 sense) and conversely that `"h:refObj"` is the inverse of `"h:backPointer"`. As a consequence the interpreter can invert the corresponding triples very easily.
  - `"h:refRel"`, `"h:refSort"`.
    The components of their value (i.e. *h:sort*, *h:position* and *h:depth*) are reached using the paths `"sort"`, `"position"` and `"depth"`.

The Caml package implementing the interpreter is designed to achieve the best performance: for instance the types built using the `SetOf` type constructor in the formal semantics (see Subsection 2.2.1 and Subsection 2.2.2) are maintained as ordered lists to optimize the set theoretic operations (i.e. `union`, `intersect`, `diff`, `sub`, `meet`, `eq`).

The package interface (shown in Subsection A.1.2) is based on a Caml functor connecting the interpreter to a generic logging device, and provides the methods shown in Figure 4.2.

The first argument is always a sting of flags (encoded as letters) specifying the operating mode (PostgreSQL or Galax) plus other settings for output verbosity and the like.

`execute` issues a query, while the other functions handle the connection between the interpreter and PostgreSQL or Galax (the methods' names are self-explanatory).

4.1.3 The HELM query generator

The query generator is the Caml package that is responsible for building specific kinds of MathQL-1 queries, which are meaningful in the context of HELM, starting from a high-level description of the wanted results. The generator is maintained by us and is currently made of three modules: two *auxiliary modules* generate the "high-level descriptions" and the *core module* generates the queries from these descriptions.

The core module of the query generator

Currently the *core module* can generate two kinds of queries:

- The *locate* query retrieves the set of HELM objects having a specified short name. The high-level description of this query is just a string containing the sort name (call it NAME-STRING) and the generated query is:

  ```
  property inverse "hth:shortname" of NAME-STRING
  ```

  This query is useful because we usually now an object by its short name (i.e. "nat" for the set of natural numbers, "plus" for the function which adds two natural numbers) and not by its HELM URI (for "plus" that is `<cic:/Coq/Init/Peano/plus.con>`). Note that many objects may share the same short name (currently there are 4 objects named "plus" and 328 objects mysteriously named "A").

- The *compose* query retrieves the set of HELM objects whose references satisfy a given set C of constraints. This query exploits the information available through *h:refObj*, *h:refRel* and *h:refSort*, which are the HELM RDF properties describing the entities that a HELM object refers to (see Subsection 3.2.4), and is build incrementally on the elements of C.

  The set C can contain 8 kinds of constrains that we list below using the numbers from 1 to 8. Furthermore each constraint can be either "positive" (we want all objects satisfying it) or "negative" (we want all objects not satisfying it).

  The following constraints take for parameters: R represents the URI of a HELM object, S represents a CIC sort, P represents a position specification and D represents

```
The basic query generated for constraint "+1" (first) and "-1" (second):
property inverse "h:refObj" istrue "occurrence" in R, "position" in P, "depth" in D of ""
property inverse "h:refObj" isfalse "occurrence" in R, "position" in P, "depth" in D of ""

The basic query generated for constraint "+2" (first) and "-2" (second):
property inverse "h:refRel" istrue "sort" in S, "position" in P, "depth" in D of ""
property inverse "h:refRel" isfalse "sort" in S, "position" in P, "depth" in D of ""

The basic query generated for constraint "+3" (first) and "-3" (second):
property inverse "h:refRel" istrue "position" in P, "depth" in D of ""
property inverse "h:refRel" isfalse "position" in P, "depth" in D of ""

The basic query generated for constraint "+4" (first) and "-4" (second):
select @obj from ... where not proj
 property "h:refObj" isfalse "occurrence" in R, "position" in P, "depth" in D of proj @obj
select @obj from ... where not proj
 property "h:refObj" istrue "occurrence" in R, "position" in P, "depth" in D of proj @obj

The basic query generated for constraint "+5" (first) and "-5" (second):
select @obj from ... where not proj
 property "h:refSort" isfalse "sort" in S, "position" in P, "depth" in D of proj @obj
select @obj from ... where not proj
 property "h:refSort" istrue "sort" in S, "position" in P, "depth" in D of proj @obj

The basic query generated for constraint "+6" (first) and "-6" (second):
select @obj from ... where not proj
 property "h:refRel" isfalse "position" in P, "depth" in D of proj @obj
select @obj from ... where not proj
 property "h:refRel" istrue "position" in P, "depth" in D of proj @obj

The basic query generated for constraint "+7" (first) and "-7" (second):
select @obj from ... where proj
 property "h:refObj" istrue "position" in P, "depth" in D of proj @obj sub R
select @obj from ... where proj
 property "h:refObj" isfalse "position" in P, "depth" in D of proj @obj sub R

Here ... is a placeholder for a query built from other constraints or for
property / of pattern ".*" (retrieving all HELM objects) if none is given.
```

Figure 4.3: The basic queries generated by the "compose" method

```
The basic query generated for constraint "+8" (first) and "-8" (second):
select @obj from ... where proj
 property "h:refSort" istrue "position" in P, "depth" in D of proj @obj sub S
select @obj from ... where proj
 property "h:refSort" isfalse "position" in P, "depth" in D of proj @obj sub S

Here ... is a placeholder for a query built from other constraints or for
property / of pattern ".*" (retrieving all HELM objects) if none is given.
```

Figure 4.4: The basic queries generated by the "compose" method (continued)

a depth index. These are the values of the properties *h:occurrence*, *h:sort*, *h:position* and *h:depth* described in Subsection 3.2.4. Note that these parameters are optional.

The query corresponding to each constraint is shown in Figure 4.3 and Figure 4.4.

±1. The wanted objects must (or must not) contain a reference to a object R in a position P with a depth index D.

±2. The wanted objects must (or must not) contain a reference to a CIC sort S in a position P with a depth index D.

±3. The wanted objects must (or must not) contain a reference to a bound variable in a position P with a depth index D.

These constraints are composed conjunctively in the sense that if the set C contains two of them, the wanted objects must satisfy both and so the corresponding basic queries are composed using the `intersect` operator. Note that each basic query returns a set of HELM URI's each without attributes (because no `attr` clause is specified in it) so the way `intersect` handles the attributes is not relevant here.

±4. The wanted objects may contain a reference to an object only if it does (or does not) concern a object R in a position P with a given depth index D.

±5. The wanted objects may contain a reference to a CIC sort only if that does (or does not) concern a sort S in a position P with a depth index D.

±6. The wanted objects may contain a reference to a bound variable only if in (or not in) a position P with a depth index D.

These constraints are composed disjunctively in the sense that if the set C contains two of them, the wanted objects must satisfy either one or the other, and no other

```
val locate  : string -> MathQL.query
val compose : spec list -> MathQL.query
```

Figure 4.5: The main methods of the generator core module

references are allowed. Note that these queries are based on negative conditions: i.e. we are seeking the objects for which the set of unwanted references is empty, therefore their `property` operators are composed using `intersect` as well.

Also note the formal symmetry between the `property` operations involved in the basic queries ±1, ±2, ±3 and ±4, ±5, ±6.

±7. The wanted objects may contain a reference to a object in a position P with a depth index D only if it does (or does not) concern an object R.

±8. The wanted objects may contain a reference to a CIC sort in a position P with a depth index D only if it does (or does not) concern a CIC sort S.

The basic queries of this kind are composed disjunctively, as the basic queries 4, 5 and 6, but using true disjunctive operators (as `Union`, `meet` or Boolean `or`) because they are based on positive conditions.

Given a *compose* query, obtained by composing the above basic blocks, we define its *static complexity* as the number of involved constraints (i.e. the cardinality of the set C). Its *dynamic complexity* is instead the number of times metadata are accessed by an interpreter executing it. Note that these values are not the same because the basic queries 4, 5, 6, 7 and 8 have a `property` operator inside a `where` clause, so the `property` operation may be iterated many times when these queries are executed.

The section about the *auxiliary modules* of the generator contains an example of how the above basic queries can be composed.

The interface of the *core module* is shown is Subsection A.1.3. The two main methods are reported in Figure 4.5 where `spec` is a type with eight constructors, one for each kind of constraint mentioned above. The parameters R, S, P and D are represented by lists of strings (like the m.s.v.'s) rather than by single strings, to allow grouping. For instance the seventh constructor (whose name is `WOnlyObj`) with a positive sign, R = {"A1", "A2"} and P = "B1", represents two grouped constraints of kind +7 with P = "B1". One of them has R = "A1" and the other has R = "A2". The constraints grouped

in a single constructor are composed disjunctively and the resulting queries (one from each constructor) are composed conjunctively. Note that the disjunctive composition is guaranteed by the ⨾ operator appearing in the semantics of the `in` part of the `istrue` and `isfalse` clauses (see the `property` operator in Subsection 2.2.2).

The auxiliary modules of the query generator

The first *auxiliary module* generates the constraints for the queries needed by the *Search-Pattern_Apply* tactic of the HELM proof engine (see Subsection 3.3.2). The main function inspects the goal the proof engine is focused on, and builds a list of constraints for the *compose* method of the *core module*. The resulting query retrieves a set of HELM statements that can possibly apply to the goal.

[GS03] reports the following example of such a query, obtained from the simple goal $2 * m \leq 2 * n$ under the assumptions m : nat and n : nat.
The query was built by the MathQL-1.2 version of the generator that could only handle constraints of kind 1 and 7 without the mention of `"depth"` because the metadata about *h:refRel*, *h:refSort*, *h:depth* and *h:sort* were computed after [GS03] was written.
The constraints derived from the analysis of the goal were the following:

A. $\leq$ (HELM URI: `<cic:/Coq/Init/Peano/le.ind#1/1>`) must appear as the main operator in the conclusion of the statement.

B. $*$ (HELM URI: `<cic:/Coq/Init/Peano/mult.con>`) must appear in the conclusion of the statement but not in main position.

C. 2 must also appear in the conclusion of the statement not in main position and 2 is encoded as the successor of the successor of zero so the above condition on 2 must hold for "successor" (HELM URI: `<cic:/Coq/Init/Datatypes/nat.ind#1/1/2>`) and "zero" (HELM URI: `<cic:/Coq/Init/Datatypes/nat.ind#1/1/1>`).

D. No other object must be referred in the conclusion of the statement,

Note that conditions A, B and C are constraints of kind 1 while condition D is a constraint of kind 7 (expressed in a negative way). So the generated query was composed by four basic queries of kind 1 and by four basic queries of kind 7. Namely the textual layout of query in [GS03] syntax is shown in Figure 4.6.

Here, `select @uri0` introduces the group of basic queries of kind 7 which are stated in its `where` clause. In this clause, `select @uri` is semantically equivalent to:

```
let $positions be {"MainConclusion", "InConclusion"} in
let $universe be
 {"cic:/Coq/Init/Datatypes/nat.ind#1/1/1", "cic:/Coq/Init/Peano/mult.con",
  "cic:/Coq/Init/Datatypes/nat.ind#1/1/2", "cic:/Coq/Init/Peano/le.ind#1/1"
 } in
 select @uri0 in
   select @uri in
    relation inverse "refObj" "cic:/Coq/Init/Peano/le.ind#1/1"
     attr $pos <- "position"
   where ex "MainConclusion" sub @uri.$pos
 intersect
   select @uri in
    relation inverse "refObj" "cic:/Coq/Init/Peano/mult.con"
     attr $pos <- "position"
   where ex "InConclusion" sub @uri.$pos
 intersect
   select @uri in
    relation inverse "refObj" "cic:/Coq/Init/Datatypes/nat.ind#1/1/2"
     attr $pos <- "position"
   where ex "InConclusion" sub @uri.$pos
 intersect
   select @uri in
    relation inverse "refObj" "cic:/Coq/Init/Datatypes/nat.ind#1/1/1"
     attr $pos <- "position"
   where ex "InConclusion" sub @uri.$pos
where
   refof select @uri in relation "refObj" refof @uri0 attr $pos <- "position"
         where ex $positions meet @uri.$pos
   sub $universe
```

Figure 4.6: Example query in the syntax of [GS03]

```
 property "h:refObj" istrue "position" in "MainConclusion" of proj @uri0
union
 property "h:refObj" istrue "position" in "InConclusion" of proj @uri0
```

in the syntax of Chapter 2, which shows its basic components composed disjunctively, but the condensed form has some optimizations:

- It uses a single `relation` operator (called `property` in Chapter 2).
- Exploits the `meet` operator for the test:

  `{"MainConclusion", "InConclusion"} meet @uri.$pos`

  which should be more efficient than the equivalent compound test:

  `"MainConclusion" sub @uri.$pos or "InConclusion" sub @uri.$pos`

  as this involves a larger number of operators.
- Uses a `$positions` variable that is defined at the beginning of the query outside the `where` clauses in place of the constant set `{"MainConclusion", "InConclusion"}`.

Finally the `$universe` variable is used in place of the constant construction:

```
subj "cic:/Coq/Init/Peano/le.ind#1/1" union subj "cic:/Coq/Init/Peano/mult.con" union
subj "cic:/Coq/Init/Datatypes/nat.ind#1/1/2" union
subj "cic:/Coq/Init/Datatypes/nat.ind#1/1/1"
```

which again shows its components composed disjunctively (each coming from a basic query of kind 7) but that is very heavy from the computational standpoint.

The `in` clause of `select @uri0` contains the four basic queries of kind 1 separated by three `intersect` operators (i.e. composed conjunctively). In each of these, a `select` operator is used because the `relation` operator of [GS03] does not have the `istrue` clause of the `property` operator described in Chapter 2.

Using the new syntax, the above query would be written as in Figure 4.7 (where we omitted the Namespace of "MainConclusion" and "InConclusion"). As we see, this version is simpler and is expected to run much faster (see Section 4.2).

The second *auxiliary module* works like the former, but inspects a closed CIC terms (i.e. not just its conclusion but its premises too). The resulting queries are used to retrieve the HELM objects having a given shape (see Subsection 4.2.3 for the solution of an advanced problem of this kind: i.e. how to find the transitive principles stored in the library).

```
let $positions be {"MainConclusion", "InConclusion"} in
let $universe be
 {"cic:/Coq/Init/Datatypes/nat.ind#1/1/1", "cic:/Coq/Init/Peano/mult.con",
  "cic:/Coq/Init/Datatypes/nat.ind#1/1/2", "cic:/Coq/Init/Peano/le.ind#1/1"
 } in
 select @obj from
   property inverse "h:refObj"
    istrue "occurrence" in "cic:/Coq/Init/Peano/le.ind#1/1",
           "position" in "MainConclusion"
    of ""
  intersect
   property inverse "h:refObj"
    istrue "occurrence" in "cic:/Coq/Init/Peano/mult.con",
           "position" in "InConclusion"
    of ""
  intersect
   property inverse "h:refObj"
    istrue "occurrence" in "cic:/Coq/Init/Datatypes/nat.ind#1/1/2",
           "position" in "InConclusion"
    of ""
  intersect
   property inverse "h:refObj"
    istrue "occurrence" in "cic:/Coq/Init/Datatypes/nat.ind#1/1/1"
           "position" in "InConclusion"
    of ""
 where
   proj property "h:refObj" istrue "position" in $positions of proj @obj
  sub $universe
```

Figure 4.7: Example query in the syntax of Chapter 2

The HELM component for the query generator

This component (also referred to as the query engine) is an http client/server, maintained by C. Sacerdoti Coen, (see Subsection 3.3.3) exporting the functionalities of both the generator and of the interpreter. Currently the query engine implements four functionalities:

- The *execute* method that takes a general MathQL-1 query and gives it to the interpreter without manipulation. In this case the input query is linearized in textual syntax inside the *execute* method URL but other solutions may be possible.
- The *locate* method that execute a *locate* query, built by the generator core module.
- The *matchConclusion* and *searchPattern* methods that execute a *compose* query built by the first and by the second auxiliary module respectively.

The *locate* method is invoked by the HELM CIC textual parser (maintained by C. Sacerdoti Coen) to translate short names into URI's because the Caml representation of a CIC term used by HELM, encodes a referred object with its URI, but the parser accepts short names too. Note that the translation procedure exploits the HELM type-checker (see Subsection 3.3.2) to disambiguate a short name referring to multiple objects.

The *matchConclusion* method is invoked by the HELM proof engine during the execution of the *SearchPattern_Apply* tactic which takes the current goal, finds a maximal set C_{max} of constraints considering all its references, queries the library for the objects satisfying these constraints, and tries to apply them to the goal. A subsets of C_{max} may also be selected in case the whole set of constraints is too restrictive. Note that the tactic pre-computes a default family of subsets of C_{max} among which the user can choose.

The execute, *locate* and *searchPattern* methods can be invoked from the proof engine graphical front-end (in this context these are regarded as additional services).

All methods can also be invoked through a Web interface[7] (maintained by A. Nediani [Ned03]), which currently understands only MathQL-1.2, but that will be updated soon.

4.1.4 The testing software

The testing software for the MathQL-1 Suite consists of a Caml Package maintained by us and providing three textual interfaces (one for the basic MathQL-1 package, one for the interpreter and one for the query generator) with specific features meant for testing.

[7]Reachable from the HELM Library homepage: `<http://helm.cs.unibo.it/library.html>`.

- The textual interface for the basic MathQL-1 package is used to test the parsers and renderers. So it just reads its input from a file (a query or a query result) and displays it. Some options can be specified in the command line to switch between textual and XML format (both for parsing and rendering).
- The textual interface for the interpreter executes a query read from a file. the string containing the options for the interpreter can be specified in the command line.
- The textual interface for the query generator can:
  - Execute general queries stored in a text or XML file.
  - Execute *locate* queries (the short names are given in the command line).
  - Execute *compose* queries reading the list of wanted constraints from a file. We defined a specific textual syntax for denoting these constraints.
  - Execute the *matchConclusion* and *searchPattern* queries produced by the generator's auxiliary modules, starting from a CIC term stored in a text file and from a progressive number identifying one of the default subsets of constraints pre-computed for that term (See Subsection 4.1.3).

  We plan to add other features according to the kinds of tests we will need to perform.

4.2 Testing the MathQL-1 Suite for HELM

In this section we present the results of the testing activity concerning the MathQL-1 Suite for HELM. Note that this testing activity is still in progress, especially the one about MathQL-1.3, which is the language presented in this dissertation.

4.2.1 The "165 queries" performance test

The most authoritative overall performance test concerning the MathQL-1.2 interpreter and query generator appears in [GS03] and was performed by us. This test reports the total execution times of 165 *matchConclusion* queries built by the *searchPattern_Apply* tactic of the HELM proof engine (see Subsection 4.1.3 and Subsection 4.1.4) starting from 33 goals and selecting each pre-computed set of constraints (the testing software provides for a specific feature for this). The goals where chosen according to these main criteria:

- They are objects of the library of sort "Prop" (i.e. propositions) whose type (i.e. statement) starts with a λ-application. This is a frequent kind of goal in practice.

Natile interpreter: PostgreSQL mode			
size	**issued queries**	**time/size (mean)**	**time/size (variance)**
1 to 2	59	0.25 sec.	0.23 sec.
3 to 9	106	0.03 sec.	0.02 sec.
1 to 9	165	0.11 sec.	0.17 sec.
Natile interpreter: Galax mode			
size	**issued queries**	**time/size (mean)**	**time/size (variance)**
1 to 2	59	13.00 sec.	13.53 sec.
3 to 9	106	0.33 sec.	0.23 sec.
1 to 9	165	4.86 sec.	10.12 sec.

Figure 4.8: Results of the "165 queries" performance test

- Their statement has at least 10 references to other objects and the height of the deepest reference in the syntactic tree of the statement is at least 5.[8]

A specific software was implemented by us in order to identify the objects of the library undergoing the above criteria for use in this test[9]. These objects are actually more than 33 (especially now that the library has been expanded) but the test is not fully automatic yet and one of its parts requires a human contribution that is proportional to the number of tested goals. We plan to eliminate this drawback as soon as possible.

The test was executed by the Natile interpreter [Nat02] running in both "PostgreSQL" and "Galax" mode. The same test performed with the Lordi interpreter [Lor02] ("PostgreSQL" mode only) is not available yet because of an error in the timing routines (the HELM team does not maintain that interpreter any more) but we plan to fix the error in order to perform the test. [GS03] reports the test results shown in Figure 4.8.

The queries explored a library of 15000 objects and were executed on an "INTEL Pentium 4" at 1.8 GHz. The PostgreSQL database was on a "AMD Athlon" at 1.5 GHz. The Galax engine could access 84 Mbytes of RDF metadata.

The "size" reported in the tables is the number of resources stored in the "$universe" variable of those queries (see Figure 4.6), which is the half of the number of basic queries

[8]This is just to make the goals reasonably complex.

[9]Unfortunately current HELM metadata do not allow to retrieve these objects with a MathQL-1 query.

Average execution time	
Natile interpreter: PostgreSQL mode	$\approx$ 105 min.
Guidi interpreter: PostgreSQL mode	$\approx$ 1 min.

Figure 4.9: Results of the "referenced objects" performance test

composing them (formally each resource gave rise to two basic queries: one of kind 1 and one of kind 7). Thus the "size" is proportional to the *static complexity* of the queries as defined in Subsection 4.1.3. Another interesting parameter of those queries would be their *dynamic complexity* but no test was done for this at the time of [GS03].

Finally note that, according to the above results, the Natile interpreter seems to handle complex queries more efficiently. We plan to enquire this fact with other tests.

4.2.2 The "referred objects" performance test

The "165 queries" performance test is not available for our interpreter yet because the related testing software is still under development, but we are sure that this interpreter runs much faster than Natile's one, because we reorganized the code completely putting the major efforts on gaining performance. The benefits of the code reorganization are clearly proven by the following test, which aims at retrieving the objects that are referred by some object of the library. This is a "heavy" test because it forces the interpreter to inspect every HELM RDF Model concerning the *h:refObj* property.

The test was performed by us on the same hardware described in Subsection 4.2.1 using the current version of the HELM library, which contains 37853 resources, and the results are shown in Figure 4.9.
The issued query, which retrieved 25955 results, was (in the syntax of Chapter 2):

```
property "h:refObj" of pattern ".*"
```

4.2.3 The "transitive principles" accuracy test

In this section we present a query that was thought by the HELM team to test the suitability of the HELM metadata model against the problem of identifying the statements of the library that are instances of a given parametric statement.

In particular the problem we want to solve is that of retrieving the HELM theorems proving that some binary relation is transitive. These theorems are of the form:

$$C : (z : A)\ (x : A)\ (y : A)\ (B\ x\ z) \to (B\ z\ y) \to (B\ x\ y)$$

where $A : \text{Set}$ and $B : A \to A \to \text{Prop}$. So the query is performed in three steps:

A. Find the objects $A : \text{Set}$.

   The statements of these objects must have no premises and a reference to Set appearing in the main position of their conclusion, so we use a basic query of kind 2 with parameters S = "Set" (the wanted sort), P = "MainConclusion" (the wanted position of the sort) and D = "0" (the depth of the reference, that in the case of a "main conclusion" is the number of premises of the statement).

B. Find the objects $B : A \to A \to \text{Prop}$.

   Here we use a similar approach as the statements of these objects must have two premises and a reference to Prop appearing in the main position of their conclusion, so we use another basic query of type 2 with parameters S = "Prop", P = "MainConclusion" and D = "2". An Additional check on each B is needed to verify that its two premises refer to the same object in main position and that this object is of type Set. We obtain this by checking that the set of objects referred by B in position *h:MainHypothesis*[10] is made of just one element contained in the set of results found in the former step.

C. Find the objects $C : (z : A)\ (x : A)\ (y : A)\ (B\ x\ z) \to (B\ z\ y) \to (B\ x\ y)$.

   The initial constraints on C are that it must have 5 premises and its conclusion must contain a reference to an object B in main position: the set of these C's is found joining, for each B found in the former step, the results of the basic queries of kind 1 with parameters R = B, P = "MainConclusion", D = "5". An Additional check on C is needed to ensure that its premises and conclusion just refer to B and A in main position (where A actually depends on B as we saw in the previous step).

   The first part of this check deals with excluding the C's whose statement refers to objects not in main positions and this is done by imposing two basic constraints of kind -4, one with the parameter P set to "InHypothesis" and the other with the parameter P set to "InConclusion", on the set of the initial C's. The second part of the test deals with selecting the above C's whose premises refer only to B and A in

[10] This set contains the *h:occurrence* components of the *h:refObj* dependencies of B.

main position and this is done by imposing two basic constraints of kind 4, one with parameters R = B, P = "MainHypothesis", and the other with parameters R = A, P = "MainHypothesis".

The resulting query is shown in Figure 4.10 and it retrieves 55 objects covering the majority of the transitive principles available in the HELM library. The best execution times (on the same hardware of the former tests) and the number of times metadata are accessed (i.e. the dynamic complexity) are shown in Figure 4.12. Note that the average execution times usually depend on the hardware workload at the moment of the query execution. Figure 4.12 confirms that our interpreter is much faster than Natile's one.

Note that the current query generator can not build such a complex query automatically, but we are planning to add more features to it, which will make it possible to obtain that query (or at least a significant part of it) from a suitable high-level description.

```
(* Preliminary commands and declarations ************************************************)
stat log keep in
let $IH be "http://www.cs.unibo.it/helm/schemas/schema-helm#InHypothesis" in
let $IC be "http://www.cs.unibo.it/helm/schemas/schema-helm#InConclusion" in
let $MH be "http://www.cs.unibo.it/helm/schemas/schema-helm#MainHypothesis" in
let $MC be "http://www.cs.unibo.it/helm/schemas/schema-helm#MainConclusion" in
let $SET be "http://www.cs.unibo.it/helm/schemas/schema-helm#Set" in
let $PROP be "http://www.cs.unibo.it/helm/schemas/schema-helm#Prop" in
(* First step ***************************************************************************)
let $sets be proj
  property inverse "h:refSort" istrue "sort" in $SET, "position" in $MC, "depth" in "0"
  of "" in
(* Second step **************************************************************************)
let %rels0 be
  for @uri in
    property inverse "h:refSort" istrue "sort" in $PROP, "position" in $MC, "depth" in "2"
    of ""
  sup
    add
      proj property "h:refObj" main "occurrence" istrue "position" in $MH of proj @uri
      as "set" (* The premises of the binary relation are saved in the "set" attribute *)
    in @uri
in
  (* Test in the second step ************************************************************)
let %rels be
  select @uri from %rels0 where ex count @uri."set" eq "1" and @uri."set" sub $sets
in
(* Third step ***************************************************************************)
let %trans0 be
  for @uri in %rels
  sup
    add (* These attributes are attached to the following "property" *)
      proj @uri as "rel",      (* The relation is saved in the "rel" attribute *)
      proj "set" @uri as "set" (* The premises of the relation are copied in "set" *)
    in
    property inverse "h:refObj" main "occurrence" istrue "position" in $MC, "depth" in "5"
    of proj @uri
in (* Continues in Figure 4.11 *)
```

Figure 4.10: The "transitive principles" query in the syntax of Chapter 2

```
let %trans1 be
  for @uri in %trans0
  sup
    add distr (* We need another couple of attributes that must go in the same group *)
      proj property "h:refObj" main "occurrence" istrue "position" in $MH of proj @uri
      as "premises", (* The premises of the trans. principle are saved in "premises"
      proj property "h:refObj" main "occurrence" istrue "position" in {$IC, $IH}
           of proj @uri
      as "extra" (* The unwanted references are saved in the "extra" attribute *)
   in @uri
in
  (* Test in the third step ************************************************************)
select @uri from %trans1
where ex not @uri."extra" and @uri."premises" sub {@uri."rel", @uri."set"}
```

Figure 4.11: The "transitive principles" query in the syntax of Chapter 2 (continued)

Best execution time		Access to the metadata base
Natile interpreter: PostgreSQL mode	92.71 sec.	Still unknown
Guidi interpreter: PostgreSQL mode	0.60 sec.	2231 times

Figure 4.12: Results of the "transitive principles" accuracy test

Chapter 5

Conclusions and future work

In this dissertation we presented a query language for RDF metadata which aims at providing the major features required by the RDF community. In particular the metadata access model used by our language exploits an arbitrary RDF Schema graph which includes property hierarchies. Moreover the language has a well defined semantics which we described using a natural operational style.

Looking at the current implementations and proposals of RDF query languages, the main novelty of our approach is the definition of a syntax for query results (with its own rigorous semantics) besides the usual syntax for queries.

This feature makes our language particularly suited for use in distributed systems where query engines are implemented as stand-alone units, because in this setting, query results must travel between the system components as well as queries, and thus both need to be encoded in a rigorously defined format (which in our case can be textual or XML).

We would like to stress that, in the spirit of the Semantic Web, distributed information bases exploiting RDF metadata are likely to become popular in the near future. For this reason we consider the presence of a specific syntax for query results as an important requirement for a modern RDF query language.

Finally note that a query result should not be just a collection of resources (as for "RDF Query") because this approach allows only queries of kind "A" in the list below:

A. The (constraints on) metadata are known and the (satisfying) resources are wanted.

B. The resources are known and the (corresponding) metadata are wanted.

In this sense an RDF query language allowing a query to return property values as well as resources (possibly placed in a structured container), would be much more desirable.

This consideration justifies the choice of using our 4-dimensional sets of *attributed values* as containers for the query results of our language.

The language we presented is being tested in the context of the HELM project where it is used to query a library of formalized mathematical knowledge. The testing activity is still in progress because the language is still unstable as well as HELM metadata architecture, and this forces the HELM team to review the query software periodically.

In the next future we plan to continue the testing and research activity concerning all aspects of the MathQL-1 Suite, with the aim of obtaining a stable software product.

The objectives of our testing and research activity are at least the following:

- Improve the test of Subsection 4.2.1 extending it to more than 200 queries.
- Use the above test to compare the performance of the three implemented interpreters (Lordi [Lor02], Natile [Nat02] and Guidi). We expect the Natile interpreter to run approximately as fast as the Lordi interpreter because we believe that changing the semantics of the language from [Lor02] to [GS03] did not affect the overall performance of a well-implemented query engine in real cases.
- Evaluate the new *constraint* feature of the `property` operator (see Subsection 2.1.2) comparing, on the Guidi interpreter, the performance of queries like:

```
select @obj from
  property inverse "h:refObj" main "occurrence" attr "position"
  of "cic:/Coq/Init/Peano/mult.con"
where ex "InConclusion" sub @obj."position"

property inverse "h:refObj" main "occurrence" istrue "position" in "InConclusion"
of "cic:/Coq/Init/Peano/mult.con"
```

  The second query should run faster because the Guidi interpreter can encode the `istrue`/`isfalse` constraints in the low-level query providing the result of `property`.
- Perform other accuracy tests on the capability of HELM metadata to describe mathematical resources to be retrieved on the basis of semantically meaningful constraints.
- Instruct the query generator to build sophisticated queries like the one of Subsection 4.2.3 starting from suitable high-level descriptions.
- Make the query interpreter completely HELM independent.
- Implement the MathQL-2 and MathQL-3 languages mentioned in Subsection 2.1.1.

Appendix A

Some Caml code from the MathQL-1 Suite

This appendix contains a selection of the most recent Caml modules composing the MathQL-1 Suite for HELM. Section A.1 includes the interfaces of the three main modules: the MathQL-1 basic module, the query interpreter and the query generator, while Section A.2 includes the most relevant implementation files. We would like to stress that all the Caml code appearing in this Appendix was personally written and tested by us.

A.1 The interfaces of the main modules

In this section we show the complete Caml interface files, concerning the MathQL-1 basic module, the interpreter and the generator, which we briefly mentioned in Section 4.1.

A.1.1 The basic MathQL-1 module: mathQL.ml

This module defines the Caml representation of MathQL-1.3 queries and query results.

```
(* output data structures ************************************************)
type path            = string list            (* the name of an attribute *)
type value           = string list            (* the value of an attribute *)
type attribute       = path * value           (* an attribute *)
type attribute_group = attribute list         (* a group of attributes *)
type attribute_set   = attribute_group list   (* the attributes of an URI *)
type av              = string * attribute_set (* an attributed URI *)
type av_set          = av list                (* the query result *)
type result = av_set
(* input data structures *************************************************)
type svar = string (* the name of a variable for an av set *)
type avar = string (* the name of a variable for an av *)
```

```
type vvar = string (* the name of a variable for an attribute value *)
type inverse = bool
type refine = RefineExact
            | RefineSub
            | RefineSuper
type main = path
type pattern = bool
type exp = path * (path option)
type exp_list = exp list
type allbut = bool
type xml = bool
type source = bool
type bin = BinFJoin (* full union - with attr handling *)
         | BinFMeet (* full intersection - with attr handling *)
         | BinFDiff (* full difference - with attr handling *)
type gen = GenFJoin (* full union - with attr handling *)
         | GenFMeet (* full intersection - with attr handling *)
type test = Xor
          | Or
          | And
          | Sub
          | Meet
          | Eq
          | Le
          | Lt
type query = Empty
           | SVar of svar
           | AVar of avar
           | Subj of msval
           | Property of inverse * refine * path *
                         main * istrue * isfalse * exp_list *
                         pattern * msval
           | Select of avar * query * msval
           | Bin of bin * query * query
           | LetSVar of svar * query * query
           | LetVVar of vvar * msval * query
           | For of gen * avar * query * query
           | Add of bool * group * query
           | If of msval * query * query
           | Log of xml * source * query
```

```
           | StatQuery of query
           | Keep of allbut * path list * query
and msval = False
          | True
          | Not of msval
          | Ex of avar list * msval
          | Test of test * msval * msval
          | Const of string
          | Set of msval list (* n-ary union *)
          | Proj of path option * query
          | Dot of avar * path
          | VVar of vvar
          | StatVal of msval
          | Count of msval
and group = Attr of (path * msval) list
          | From of avar
and con = pattern * path * msval
and istrue = con list
and isfalse = con list
```

A.1.2 The query interpreter interface: mQueryInterpreter.mli

This file defines the interface of our MathQL-1.3 compliant interpreter.

```
module type Callbacks =
   sig
      val log : string -> unit (* logging function *)
   end
module Make (C: Callbacks) :
   sig
      val postgres : string
      val galax    : string
      val stat     : string (* shows some query statistics *)
      val quiet    : string
      val warn     : string
      val execute  : string -> MathQL.query -> MathQL.result
      val init     : string -> unit
      val close    : string -> unit
      val check    : string -> bool
   end
```

A.1.3 The query generator interface: mQueryGenerator.mli

This file defines the interface of our MathQL-1.3 compliant query generator.

```
type uri = string
type pos = string
type depth = string
type sort = string
type neg = bool
(* "Only" constraints on "Obj" and "Sort" can be "Strong" or "Weak" *)
type spec = MustObj   of neg * uri list * pos list * depth list  (* kind 1 *)
          | MustSort  of neg * sort list * pos list * depth list (* kind 2 *)
          | MustRel   of neg * pos list * depth list             (* kind 3 *)
          | SOnlyObj  of neg * uri list * pos list * depth list  (* kind 4 *)
          | SOnlySort of neg * sort list * pos list * depth list (* kind 5 *)
          | OnlyRel   of neg * pos list * depth list             (* kind 6 *)
          | WOnlyObj  of neg * uri list * pos list * depth list  (* kind 7 *)
          | WOnlySort of neg * sort list * pos list * depth list (* kind 8 *)
val locate  : string -> MathQL.query
val compose : spec list -> MathQL.query
val builtin : MathQL.vvar -> string
```

A.2 Some complete modules

This section contains a selection of complete Caml modules (consisting of an interface file and of an implementation file) that are part of the MathQL-1 Suite for HELM.

A.2.1 The query interpreter utility module: MQIUtil

This module implements the basic functionalities used by the interpreter such as the set-theoretic operations and the operations for composing a.v. sets.

Interface: mQIUtil.mli

```
val mql_true      : MathQL.value
val mql_false     : MathQL.value
val set_sub       : MathQL.value -> MathQL.value -> MathQL.value
val set_meet      : MathQL.value -> MathQL.value -> MathQL.value
val set_eq        : MathQL.value -> MathQL.value -> MathQL.value
val set_union     : 'a list -> 'a list -> 'a list
val mql_union     : ('a * 'b list) list -> ('a * 'b list) list ->
```

```
                    ('a * 'b list) list
val mql_prod      : MathQL.attribute_set -> MathQL.attribute_set ->
                    MathQL.attribute_set
val mql_intersect : MathQL.result -> MathQL.result -> MathQL.result
val mql_diff      : MathQL.result -> MathQL.result -> MathQL.result
val iter          : ('a -> 'b list) -> 'a list -> 'b list
val mql_iter      : ('c -> ('a * 'b list) list) -> 'c list ->
                    ('a * 'b list) list
val mql_iter2     : ('c -> 'd -> ('a * 'b list) list) -> 'c list ->
                    'd list -> ('a * 'b list) list
val xor           : MathQL.value -> MathQL.value -> MathQL.value
val le            : MathQL.value -> MathQL.value -> MathQL.value
val lt            : MathQL.value -> MathQL.value -> MathQL.value
val set           : string * 'a -> (string * 'a) list -> (string * 'a) list
```

Implementation: mQIUtil.ml

```
(* Boolean constants  ****************************************************)
let mql_false = []
let mql_true = [""]
(* set theoretic operations **********************************************)
let rec set_sub v1 v2 =
   match v1, v2 with
      | [], _                          -> mql_true
      | _, []                          -> mql_false
      | h1 :: _, h2 :: _ when h1 < h2  -> mql_false
      | h1 :: _, h2 :: t2 when h1 > h2 -> set_sub v1 t2
      | _ :: t1, _ :: t2               -> set_sub t1 t2
let rec set_meet v1 v2 =
   match v1, v2 with
      | [], _                          -> mql_false
      | _, []                          -> mql_false
      | h1 :: t1, h2 :: _ when h1 < h2 -> set_meet t1 v2
      | h1 :: _, h2 :: t2 when h1 > h2 -> set_meet v1 t2
      | _, _                           -> mql_true
let set_eq v1 v2 =
   if v1 = v2 then mql_true else mql_false
let rec set_union v1 v2 =
   match v1, v2 with
      | [], v                           -> v
```

```
      | v, []                          -> v (* (zz h1 ">" h2); *)
      | h1 :: t1, h2 :: t2 when h1 < h2 -> h1 :: set_union t1 v2
      | h1 :: t1, h2 :: t2 when h1 > h2 -> h2 :: set_union v1 t2
      | h1 :: t1, _ :: t2              -> h1 :: set_union t1 t2
let rec iter f = function
   | []           -> []
   | head :: tail -> set_union (f head) (iter f tail)
(* MathQL specific set operations  ****************************************)
let rec mql_union s1 s2 =
   match s1, s2 with
      | [], s                                    -> s
      | s, []                                    -> s
      | (r1, g1) :: t1, (r2, _) :: _ when r1 < r2 ->
         (r1, g1) :: mql_union t1 s2
      | (r1, _) :: _, (r2, g2) :: t2 when r1 > r2 ->
         (r2, g2) :: mql_union s1 t2
      | (r1, g1) :: t1, (_, g2) :: t2            ->
         (r1, set_union g1 g2) :: mql_union t1 t2
let rec mql_iter f = function
   | []           -> []
   | head :: tail -> mql_union (f head) (mql_iter f tail)
let rec mql_iter2 f l1 l2 = match l1, l2 with
   | [], []             -> []
   | h1 :: t1, h2 :: t2 -> mql_union (f h1 h2) (mql_iter2 f t1 t2)
   | _                  -> raise (Invalid_argument "mql_itwr2")
let rec mql_prod g1 g2 =
   let mql_prod_aux a = iter (fun h -> [mql_union a h]) g2 in
   iter mql_prod_aux g1
let rec mql_intersect s1 s2 =
   match s1, s2 with
      | [], s                                    -> []
      | s, []                                    -> []
      | (r1, _) :: t1, (r2, _) :: _ when r1 < r2 -> mql_intersect t1 s2
      | (r1, _) :: _, (r2, _) :: t2 when r1 > r2 -> mql_intersect s1 t2
      | (r1, g1) :: t1, (_, g2) :: t2            ->
         (r1, mql_prod g1 g2) :: mql_intersect t1 t2
let rec mql_diff s1 s2 =
   match s1, s2 with
      | [], _                                    -> []
      | s, []                                    -> s
```

```
      | (r1, g1) :: t1 , (r2, _) ::_ when r1 < r2 ->
         (r1, g1) :: (mql_diff t1 s2)
      | (r1, _) :: _, (r2, _) :: t2 when r1 > r2  -> mql_diff s1 t2
      | _ :: t1, _ :: t2                          -> mql_diff t1 t2
(* logic operations  ******************************************************)
let xor v1 v2 =
   let b = v1 <> mql_false in
   if b && v2 <> mql_false then mql_false else
   if b then v1 else v2
(* numeric operations  ****************************************************)
let int_of_list = function
   | [s] -> int_of_string s
   | _   -> raise (Failure "int_of_list")
let le v1 v2 =
   try if int_of_list v1 <= int_of_list v2 then mql_true else mql_false
   with _ -> mql_false
let lt v1 v2 =
   try if int_of_list v1 < int_of_list v2 then mql_true else mql_false
   with _ -> mql_false
(* context handling  ******************************************************)
let rec set ap = function
   | []                                  -> [ap]
   | head :: tail when fst head = fst ap -> ap :: tail
   | head :: tail                        -> head :: set ap tail
```

A.2.2 The query interpreter main module: MQIExecute

This module implements the function that executes a query. Its arguments are the logging function and the string holding the interpreter options.

Interface: mQIExecute.mli

```
val execute : (string -> unit) -> string -> MathQL.query -> MathQL.result
```

Implementation: mQIExecute.ml

```
(* modifiers  *************************************************************)
let galax_char = 'G'
let stat_char = 'S'
let warn_char = 'W'
(* contexts ***************************************************************)
```

```
type svar_context = (MathQL.svar * MathQL.av_set) list
type avar_context = (MathQL.avar * MathQL.av) list
type group_context = (MathQL.avar * MathQL.attribute_group) list
type vvar_context = (MathQL.vvar * MathQL.value) list
type context = {svars: svar_context;
                avars: avar_context;
                groups: group_context; (* auxiliary context *)
                vvars: vvar_context
               }
(* execute  ****************************************************************)
exception Found
let execute out m x =
   let module M = MathQL in
   let module P = MQueryUtil in
   let module U = MQIUtil in
   let warn q =
     if String.contains m warn_char then
     begin
        out "MQIExecute: warning: reference to undefined variables: ";
        P.text_of_query out q "\n"
     end
   in
   let rec eval_val c = function
      | M.False -> U.mql_false
      | M.True -> U.mql_true
      | M.Const s -> [s]
      | M.Set l -> U.iter (eval_val c) l
      | M.Test k y1 y2 ->
         let cand y1 y2 =
            if eval_val c y1 = U.mql_false then U.mql_false else eval_val c y2
         in
         let cor y1 y2 =
            let v1 = eval_val c y1 in
            if v1 = U.mql_false then eval_val c y2 else v1
         in
         let h f y1 y2 = f (eval_val c y1) (eval_val c y2) in
         let f = match k with
            | M.And  -> cand
            | M.Or   -> cor
            | M.Xor  -> h U.xor
```

```
      | M.Sub  -> h U.set_sub
      | M.Meet -> h U.set_meet
      | M.Eq   -> h U.set_eq
      | M.Le   -> h U.le
      | M.Lt   -> h U.lt
   in
   f y1 y2
| M.Not y ->
   if eval_val c y = U.mql_false then U.mql_true else U.mql_false
| M.VVar i -> begin
   try List.assoc i c.vvars
   with Not_found -> warn (M.Subj (M.VVar i)); [] end
| M.Dot i p -> begin
   try List.assoc p (List.assoc i c.groups)
   with Not_found -> warn (M.Subj (M.Dot i p)); [] end
| M.Proj None x -> List.map (fun (r, _) -> r) (eval_query c x)
| M.Proj (Some p) x ->
   let proj_group_aux (q, v) = if q = p then v else [] in
   let proj_group a = U.iter proj_group_aux a in
   let proj_set (_, g) = U.iter proj_group g in
   U.iter proj_set (eval_query c x)
| M.Ex l y ->
   let rec ex_aux h = function
      | []        ->
         let d = {c with groups = h} in
         if eval_val d y = U.mql_false then () else raise Found
      | i :: tail ->
         begin
            try
               let (_, a) = List.assoc i c.avars in
               let rec add_group = function
                  | []     -> ()
                  | g :: t -> ex_aux ((i, g) :: h) tail; add_group t
               in
               add_group a
            with Not_found -> ()
         end
   in
   (try ex_aux [] l; U.mql_false with Found -> U.mql_true)
| M.StatVal y ->
```

```
      let t0 = Sys.time () in
      let r = (eval_val c y) in
      let t1 = Sys.time () in
      out (Printf.sprintf "Stat: %.2fs,%i\n" (t1 -. t0) (List.length r));
      r
  | M.Count y -> [string_of_int (List.length (eval_val c y))]
and eval_query c = function
  | M.Empty -> []
  | M.Subj x ->
      List.map (fun s -> (s, [])) (eval_val c x)
  | M.Log _ b x ->
      if b then begin
        let t0 = Sys.time () in
        P.text_of_query out x "\n";
        let t1 = Sys.time () in
        if String.contains m stat_char then
          out (Printf.sprintf "Log source: %.2fs\n" (t1 -. t0));
          eval_query c x
      end else begin
        let s = (eval_query c x) in
        let t0 = Sys.time () in
        P.text_of_result out s "\n";
        let t1 = Sys.time () in
        if String.contains m stat_char then
          out (Printf.sprintf "Log: %.2fs\n" (t1 -. t0));
        s
      end
  | M.If y x1 x2 ->
      if (eval_val c y) = U.mql_false
        then (eval_query c x2) else (eval_query c x1)
  | M.Bin k x1 x2 ->
      let f = match k with
        | M.BinFJoin -> U.mql_union
        | M.BinFMeet -> U.mql_intersect
        | M.BinFDiff -> U.mql_diff
      in
      f (eval_query c x1) (eval_query c x2)
  | M.SVar i -> begin
      try List.assoc i c.svars
      with Not_found -> warn (M.SVar i); [] end
```

```
| M.AVar i -> begin
   try [List.assoc i c.avars]
   with Not_found -> warn (M.AVar i); [] end
| M.LetSVar i x1 x2 ->
   let d = {c with svars = U.set (i, eval_query c x1) c.svars} in
   eval_query d x2
| M.LetVVar i y x ->
   let d = {c with vvars = U.set (i, eval_val c y) c.vvars} in
   eval_query d x
| M.For k i x1 x2 ->
   let f = match k with
      | M.GenFJoin -> U.mql_union
      | M.GenFMeet -> U.mql_intersect
   in
   let rec for_aux = function
      | []     -> []
      | h :: t ->
         let d = {c with avars = U.set (i, h) c.avars} in
         f (eval_query d x2) (for_aux t)
   in
   for_aux (eval_query c x1)
| M.Add b z x ->
   let f = if b then U.mql_prod else U.set_union in
   let g a s = (fst a, f (snd a) (eval_grp c z)) :: s in
   List.fold_right g (eval_query c x) []
| M.Property q0 q1 q2 mc ct cf el pat y ->
   let f = if String.contains m galax_char
      then MQIProperty.galax else MQIProperty.postgres in
   let eval_cons (pat, p, y) = (pat, p, eval_val c y) in
   let cons_true = List.map eval_cons ct in
   let cons_false = List.map eval_cons cf in
   let property_aux s =
      let t0 = Sys.time () in
      let r = f q0 q1 q2 mc cons_true cons_false el pat s in
      let t1 = Sys.time () in
      if String.contains m stat_char then
          out (Printf.sprintf "Property: %.2fs,%i\n" (t1 -. t0) (List.length r));
      r
   in U.mql_iter property_aux (eval_val c y)
| M.StatQuery x ->
```

```
      let t0 = Sys.time () in
      let r = (eval_query c x) in
      let t1 = Sys.time () in
      out (Printf.sprintf "Stat: %.2fs,%i\n" (t1 -. t0) (List.length r));
      r
   | M.Select i x y ->
      let rec select_aux = function
         | []     -> []
         | h :: t ->
            let d = {c with avars = U.set (i, h) c.avars} in
            if eval_val d y = U.mql_false
               then select_aux t else h :: select_aux t
      in
      select_aux (eval_query c x)
   | M.Keep b l x ->
      let keep_path (p, v) t =
         if List.mem p l = b then t else (p, v) :: t in
      let keep_grp a = List.fold_right keep_path a [] in
      let keep_set a g =
         let kg = keep_grp a in
         if kg = [] then g else kg :: g
      in
      let keep_av (s, g) = (s, List.fold_right keep_set g []) in
      List.map keep_av (eval_query c x)
and eval_grp c = function
   | M.Attr l ->
      let attr_aux g (p, y) = U.mql_union g [(p, eval_val c y)] in
      [List.fold_left attr_aux [] l]
   | M.From i ->
      try snd (List.assoc i c.avars)
      with Not_found -> warn (M.AVar i); []
in
let c = {svars = []; avars = []; groups = []; vvars = []} in
let t0 = Sys.time () in
let r = eval_query c x in
let t1 = Sys.time () in
if String.contains m stat_char then
   out (Printf.sprintf "MQIExecute: %.2fs,%s\n" (t1 -. t0) m);
r
```

A.2.3 The query generator core module: MQueryGenerator

This module implements the core part of the query generator. The interface of this module is shown in Subsection A.1.3.

Implementation: mQueryGenerator.ml

```
type uri = string
type pos = string
type depth = string
type sort = string
type neg = bool
(* "Only" constraints on "Obj" and "Sort" can be "Strong" or "Weak" *)
type spec = MustObj   of neg * uri list * pos list * depth list
          | MustSort  of neg * sort list * pos list * depth list
          | MustRel   of neg * pos list * depth list
          | SOnlyObj  of neg * uri list * pos list * depth list
          | SOnlySort of neg * sort list * pos list * depth list
          | OnlyRel   of neg * pos list * depth list
          | WOnlyObj  of neg * uri list * pos list * depth list
          | WOnlySort of neg * sort list * pos list * depth list
module M = MathQL
let locate s =
   let query =
      M.Property true M.RefineExact ["objectName"] [] [] [] []
                 false (M.Const s)
   in M.StatQuery query
let compose cl =
   let letin = ref [] in
   let must = ref [] in
   let sonly = ref [] in
   let wonly = ref [] in
   let only = ref false in
   let set_val = function
      | [s] -> M.Const s
      | l   ->
         let msval = M.Set (List.map (fun s -> M.Const s) l) in
         if ! only then begin
            let vvar = "val" ^ string_of_int (List.length ! letin) in
            letin := (vvar, msval) :: ! letin;
```

```
          M.VVar vvar
       end else msval
in
let property inv n neg r s p d =
   let cons r s p d =
      let con p = function
         | [] -> []
         | l  -> [(false, [p], set_val l)]
      in
      con "occurrence" r @ con "sort" s @ con "position" p @ con "depth" d
   in
   let m = match inv ,n with
      | false, "h:refObj"  -> ["occurrence"]
      | false, "h:refSort" -> ["sort"]
      | _, _               -> []
   in
   let ct, cf = if neg then [], cons r s p d else cons r s p d, [] in
   M.Property inv M.RefineExact [n] m ct cf [] false
              (if inv then M.Const "" else M.Proj None (M.AVar "obj"))
in
let rec aux = function
   | [] -> ()
   | MustObj neg r p d :: tail ->
      only := false;
      must := property true "h:refObj" neg r [] p d :: ! must; aux tail
   | MustSort neg s p d :: tail ->
      only := false;
      must := property true "h:refSort" neg [] s p d :: ! must; aux tail
   | MustRel neg p d :: tail ->
      only := false;
      must := property true "h:refRel" neg [] [] p d :: ! must; aux tail
   | SOnlyObj neg r p d :: tail ->
      only := true;
      sonly := property false "h:refObj" (not neg) r [] p d :: ! sonly;
      aux tail
   | SOnlySort neg s p d :: tail ->
      only := true;
      sonly := property false "h:refSort" (not neg) [] s p d :: ! sonly;
      aux tail
   | OnlyRel neg p d :: tail ->
```

```
      only := true;
      sonly := property false "h:refRel" (not neg) [] [] p d :: ! sonly;
      aux tail
   | WOnlyObj neg r p d :: tail ->
      only := true;
      wonly := (property false "h:refObj" neg [] [] p d, set_val r) :: ! wonly;
      aux tail
   | WOnlySort neg s p d :: tail ->
      only := true;
      wonly := (property false "h:refSort" neg [] [] p d, set_val s) :: ! wonly;
      aux tail
in
let query = aux cl in
let rec iter f g = function
   | []           -> raise (Failure "MQueryGenerator.iter")
   | [head]       -> (f head)
   | head :: tail -> let t = (iter f g tail) in g (f head) t
in
let must_query =
   if ! must = [] then
      M.Property false M.RefineExact [] [] [] [] [] true (M.Const ".*")
   else
      iter (fun x -> x) (fun x y -> M.Bin M.BinFMeet x y) ! must
in
let sonly_val () =
   let f x = M.Proj None x in
   let g x y = M.Test M.Or x y in
   M.Not (iter f g ! sonly)
in
let wonly_val () =
   let f (x, v) = M.Test M.Sub (M.Proj None x) v in
   let g x y = M.Test M.And x y in
   iter f g ! wonly
in
let select_query x =
   match !sonly, !wonly with
      | [], [] -> x
      | _, []  -> M.Select "obj" x (sonly_val ())
      | [], _  -> M.Select "obj" x (wonly_val ())
      | _, _   -> M.Select "obj" x (M.Test M.And (sonly_val ()) (wonly_val ()))
```

```
   in
   let letin_query =
      if ! letin = [] then fun x -> x
      else
        let f (vvar, msval) x = M.LetVVar vvar msval x in
        iter f (fun x y z -> x (y z)) ! letin
   in
   M.StatQuery (letin_query (select_query must_query))
let builtin s =
   let ns = "http://www.cs.unibo.it/helm/schemas/schema-helm#" in
   match s with
      | "MH"   -> ns ^ "MainHypothesis"
      | "IH"   -> ns ^ "InHypothesis"
      | "MC"   -> ns ^ "MainConclusion"
      | "IC"   -> ns ^ "InConclusion"
      | "IB"   -> ns ^ "InBody"
      | "SET"  -> ns ^ "Set"
      | "PROP" -> ns ^ "Prop"
      | "TYPE" -> ns ^ "Type"
      | _      -> raise (Failure "MQueryGenerator.builtin")
```

References

[ABCK01] Allsopp D., Beautement P., Carson J. and Kirton M. *Toward semantic interoperability In agent-based coalition command systems*. presented at the International Semantic Web Working Symposium 2001.
<http://www.semanticweb.org/SWWS/program/full/paper10.pdf>.

[Algae] Prud'hommeaux E. *W3C Algae HOWTO*. 1999.
<http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html>.

[Ano94] Anonymous. *The QED Manifesto*. In Automated Deduction - CADE 12, volume 814 of Lecture Notes in Artificial Intelligence, pp. 238-251. Springer-Verlag, 1994.

[APSGS01] Asperti A,, Padovani L., Sacerdoti Coen C., Guidi F. and Schena I. *Mathematical Knowledge Management in HELM*. 1st International Workshop on Mathematical Knowledge Management, RISC-Linz, Austria, September 2001. To appear in Annals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management, Cluwer Academic Publishers.

[APSSa] Asperti A., Padovani L., Sacerdoti Coen C. and Schena I. *Content Centric Logical Environments*. Short Presentation at LICS 2000, Santa Barbara, California, USA, June 2000.

[APSSb] Asperti A., Padovani L., Sacerdoti Coen C. and Schena I. *Towards a Library of Formal Mathematics*. Technical Report of TPHOLs 2000 Conference, Portland, Oregon, USA, August 2000.

[APSSc] Asperti A., Padovani L., Sacerdoti Coen C. and Schena I. *Formal Mathematics in MathML*. Session Presentation at the 1st MathML International Conference, University of Illinois, Urbana-Champaign, Illinois, USA, October 2000.

[APSSd] Asperti A., Padovani L., Sacerdoti Coen C., and Schena I. *Formal Mathematics on the Web*. In Proc. of the 8th International Conference "Crimea 2001", Vol. 1, pp. 342-346, Sudak, Ukraine, June 2001.

[APSSe] Asperti A., Padovani L., Sacerdoti Coen C. and Schena I. *XML, Stylesheets and the re-mathematization of formal content*. In Proc. of Extreme Markup Languages 2001, pp. 17-27, Montréal, Québec, Canada, August 2001.

[APSSf] Asperti A., Padovani L., Sacerdoti Coen C. and Schena I. *HELM and the Semantic Math-Web*. In Proc. of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs), LNCS 2152, pp. 59-74, Edinburgh, Scotland, September 2001.

[BB99] Bosak J. and Bray T. *XML and the Second-Generation Web*. In The Eighth International World Wide Web Conference (WWW8), Scientific American, 1999.

[Bee98] Beech D. *Position Paper on Query Languages for the Web*, presented at the W3C Query Languages'98 workshop (QL'98), 1998.

[Ber98] Berners-Lee T. *What the Semantic Web can represent*. 1998 `<http://www.w3.org/DesignIssues/RDFnot.html>`.

[BHL01] Berners-Lee T., Hendler J. and Lassila O. *The Semantic Web*. Scientific American, (284(5)):34-43, 2001.

[BS02] Benzmüller C. and Sorge V. *Ω-ANTS - an open approach at combining interactive and automated theorem proving*. In Proc. of the Calculemus Symposium 2002.

[Caml] Leory X. *The Objective Caml system, release 3.06. Documentation and user's manual*. INRIA, August 19, 2002. `<http://caml.inria.fr/ocaml/htmlman/>`.

[CH88] Coquand T. and Huet G. *The Calculus of Constructions*. In Information and Computation, number 76(2/3). 1988.

[CKT95] Coscoy Y., Kahn G., and Thery L. *Extracting Text from Proofs*. In Dezani M. and Plotkin G., editors, Typed Lambda Calculus and Applications, volume 902, pp. 109-123. Springer-Verlag, 1995.

[Coq] The Coq Development Team. *The Coq Proof Assistant. Reference Manual. Version 7.4*. LogiCal Project, 2003. `<http://coq.inria.fr/doc/main.html>`.

[Com98] *Communication of the ACM*. Special Issue on Digital Libraries. 1998.

[Cos00] Coscoy Y. *Explication textuelle de preuves pour le calcul des constructions inductives*. Ph.D. dissertation, Sophia Antipolis, Université de Nice, 2000.

[CP02] Casarini P. and Padovani L. *The Gnome DOM Engine*. Markup Languages: Theory & Practice, Vol. 3, Issue 2, pp. 173-190, ISSN 1099-6621, MIT Press, April 2002.

[dBr80] De Bruijn N. G. *A survey of the project AUTOMATH*. In J. P. Seldin and J. R. Hindley, editors, To H. B. Curry: *Essays in Combinatory Logic, Lambda Calculus and Formalism*. pp. 589-606. Academic Press, 1980.

[DBSA98] Decker S., Brickley D, Saarela J. and Angele j. *A Query and Inference Service for RDF*, presented at the W3C Query Languages'98 workshop (QL'98). 1998.

[DC] *Dublin Core Metadata Element Set, Version 1.1: Reference Description*. DCMI Recommendation, February 4, 2003.
`<http://dublincore.org/documents/2003/02/04/dces/>`.

[DCT] *DCMI Metadata Terms*. DCMI Recommendation, February 12, 2003.
`<http://dublincore.org/documents/2003/02/12/dcmi-terms/>`.

[EDOS00] Ennser L., Delporte C., Oba M. and Sunil K. M. *Integrating XML with DB2 XML Extender and DB2 Text Extender*. Redbooks, 2000.

[GLT89] Girard J. Y., Lafont Y. and Taylor P. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[GM93] Gordon M. and Melham T. *Introduction to HOL: A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.

[GS03] Guidi F. and Schena I. *A Query Language for a Metadata Framework about Mathematical Resources*. In Proc. of 2nd International Conference on Mathematical Knowledge Management (MKM 2003). Bertinoro, Italy, February 2003. LNCS 2594, pp. 105-118, Springer. 2003.

[He91] Huet G. and Plotkin G. (eds). *Logical Frameworks*. Cambridge University Press, 1991.

[He93] Huet G. and Plotkin G. (eds). *Logical Environments*. Cambridge University Press, 1993.

[HP02] Hawke S. and Prud'hommeaux E. *RDF Database Access Protocol*. 2002. <http://www.w3.org/2002/01/rdf-databases/protocol>.

[ISO8879] *Standard Generalized Markup Language (SGML)*. ISO 8879:1986.

[Jut94] Jutting van L. S. B. *Checking Landau's "Grundlagen" in the AUTOMATH System*. Ph.D. thesis, Eindhoven University of Technology, 1994. Useful summary in Nederpelt, Geuvers and de Vrijier, 1994, pp. 701-732.

[Kar00] Karvounarakis G. *RDF Query Languages: a state-of-the-art*. 2000. <http://139.91.183.30:9090/RDF/publications/state.html>.

[KBN99] Kamareddine F., Bloo B. and Nederpelt R. *On* Π*-conversion in The lambda-cube and the combination with abbreviations*. In Annal of Pure and Applied Logics volume 97, no. 1-3, pp. 27-45, 1999, Elsevier, North-Holland.

[KCP+01] Karvounarakis G., Christophides V., Plexousakis D., Alexaki S. and Tolle K. *The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases*. 2001. 4th International Workshop on the Web and Databases (WebDB 2001).

[KCPA00] Karvounarakis G., Christophides V., Plexousakis D. and Alexaki S. *Querying Community Web Portals, 2000*. <http://139.91.183.30:9090/RDF/publications/sigmod2000.html>.

[KKPS01] Kahan J., Koivunen M., Prud'hommeaux E. and Swick R. R. *Annotea: An Open RDF Infrastructure for Shared Web Annotations*. In The Tenth International World Wide Web Conference (WWW10), 2001.

[KN96] Kamareddine F. and Nederpelt R. *Canonical Typing and* π*-conversion in the Barendregt Cube*. Journal of Functional Programming volume 6, no. (2), pp. 245-267, 1996, Cambridge University Press.

[Koh00a] Kohlhase M. *OMDoc: An Infrastructure for OpenMath Content Dictionary Information*. 2000. <http://www.mathweb.org/omdoc>.

[Koh00b] Kohlhase M. *OMDoc: Towards an Internet Standard for the Administration, Distribution and Teaching of mathematical Knowledge*. In Artificial Intelligence and Symbolic Computation, LNAI. Springer-Verlag, 2000.

[Koh00c] Kohlhase M. *OMDoc: Towards an OpenMath Representation of Mathematical Documents*. Technical Report, 2000. <http://www.mathweb.org/omdoc/>.

[Lan98] Laneve C. *La descrizione operazionale dei linguaggi di programmazione. Un'introduzione* FrancoAngeli, 1998.

[Lor02] Lordi D. *Sperimentazione e Sviluppo di Strumenti per la gestione di metadati.* Master Thesis in Computer Science, University of Bologna, 2002. Advisor: A. Asperti.

[LP92] Luo Z. and Pollack R. *LEGO Proof Development System: User's Manual.* 1992. <http://www.dcs.ed.ac.uk/home/lego/>.

[Mac95] MacKenzie D. *The automation of proof: A historical and sociological exploration.* In IEEE: Annals of the History of Computing, 1995.

[MathML] *Mathematical Markup Language (MathML) 2.0.* W3C Recommendation. February 21, 2001. <http://www.w3.org/TR/MathML2/>.

[Mil98] Miller E. *An Introduction to the Resource Description Framework.* D-Lib Magazine, ISSN 1082-9873, 1998.

[MT01] Miner R. and Topping P. *Math on the Web: A Status Report.* January 2001. <http://www.dessci.com/webmath/status>.

[Nat02] Natile L. *Tecnologie per l'Interrogazione di Basi Documentarie in Formato XML.* Master Thesis in Computer Science, University of Bologna, 2002. Advisor: A. Asperti.

[Ned03] Nediani A. *Disegno e Implementazione di un'Interfaccia Web di Supporto ad Interrogazioni su Basi di Dati Documentarie.* Master Thesis in Computer Science, University of Bologna, 2003. Advisor: A. Asperti.

[OpenMath] OpenMath Consortium. *The OpenMath Standard: OpenMath Deliverable 1.3.3a.* 1999. <http://www.openmath.org/>.

[PICS] *PICS Label Distribution Label Syntax and Communication Protocols.* W3C Recommendation. October 31, 1996. <http://www.w3.org/TR/REC-PICS-labels>.

[RDF] *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation. February 22, 1999. <http://www.w3.org/TR/1999/REC-rdfsyntax-19990222/>.

[RDFMT] *RDF Semantics.* W3C Working Draft. January 23, 2003. <http://www.w3c.org/TR/rdf-mt/>.

[RDFS] *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Working Draft. January 23, 2003 <http://www.w3.org/TR/rdf-schema/>.

[RDFTC] *RDF Test Cases*, W3C Working Draft. January 23, 2003. <http://www.w3.org/TR/rdf-testcases/>.

[Ric99] Ricci A. *Studio e progettazione di un modello RDF per biblioteche matematiche elettroniche*. 1999.

[Rob65] Robinson J. A. *A machine-oriented logic based on the resolution principle*. In Journal of the ACM, volume 2, pp. 23-41, 1965.

[ROSS99] Rushby J.M., Owre S., Shankar N. and Stringer-Calvert. *PVS System Guide,* 1999. Computer Science Laboratory, SRI International.

[Rud92] Rudnicki P. *An overview of the Mizar project*. In Proceedings of 1992 Workshop on Types and Proofs for Programs, pp. 311-332, 1992.

[Sam00] Sambin G. *Formal topology and domains*. Electronic Notes in Theoretical Computer Science, (35). 2000.

[Sch02] Schena I. *Towards a Semantic Web for Formal Mathematics*. Ph.D. dissertation. University of Bologna, 2002. Advisor: A. Asperti.

[TR93] Trybulec A. and Rudnicki P. *Using Mizar in Computer Aided Instruction of Mathematics*. 1993, Norwegian-French Conference of CAI in Mathematics.

[Unicode] Unicode Consortium. *The Unicode Standard, Version 3.2*. March 2002. <http://www.unicode.org/unicode/standard/standard.html>.

[URI] *Uniform Resource Identifiers (URI): Generic Syntax (RFC 2396)*. August 1998. <http://www.ietf.org/rfc/rfc2396.txt>.

[URL] *Relative Uniform Resource Locators (RFC 1808)*. June 1995. <http://www.ietf.org/rfc/rfc1808.txt>.

[W3Ca] *Character Model for the World Wide Web 1.0*, W3C Working Draft. April 30, 2002. <http://www.w3.org/TR/charmod/>.

[W3Cb] *Web Ontology Language (OWL) Use Cases and Requirements*. W3C Working Draft. February 3, 2003. <http://www.w3.org/TR/webont-req/>.

[Wan60] Wang H. *Toward mechanical mathematics*. In IBM Journal of research and development, volume 4, pp. 2-22, 1960.

[Win93] Winskel G. *The formal semantics of programming languages: an introduction*. MIT Press Series in the Foundations of Computing. London: MIT Press, 1993.

[XHTML] *XHTML[tm] 1.0 The Extensible HyperText Markup Language (Second Edition). A Reformulation of HTML 4 in XML 1.0*. W3C Recommendation. January 26, 2000, revised August 1, 2002. <http://www.w3.org/TR/xhtml1/>.

[XLink] *XML Linking Language (XLink) 1.0*. W3C Recommendation. June 27, 2001. <http://www.w3.org/TR/xlink>.

[XML] *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation. October 6, 2000. <http://www.w3.org/REC-xml>.

[XMLN] *Namespaces in XML*. W3C Recommendation. January 14, 1999. <http://www.w3.org/TR/REC-xml-names/>.

[XMLS] *XML Schema*. W3C Recommendation. May 2, 2001.
Primer: <http://www.w3.org/TR/xmlschema-0/>,
Structures: <http://www.w3.org/TR/xmlschema-1/>,
Datatypes: <http://www.w3.org/TR/xmlschema-2/>.

[XPointer] *XML Pointer Language (XPointer)*. W3C Working Draft. August 16, 2002 <http://www.w3.org/TR/xptr>.

[XQuery] *XQuery 1.0: An XML Query Language*. W3C Working Draft November 15, 2002. <http://www.w3.org/TR/xquery/>.

[XSLT] *XSL Transformations (XSLT) 1.0*. W3C Recommendation. November 16, 1999. <http://www.w3.org/TR/xslt>.

[Zac03] Zacchiroli S. *Web services per il supporto alla dimostrazione interattiva*. Master Thesis in Computer Science, University of Bologna, 2003. Advisor: A. Asperti.